

# Unix System Manager's Manual



Printed by the USENIX Association as a service to the UNIX Community. This material is copyrighted by The Regents of the University of California and/or Bell Telephone Laboratories, and is reprinted by permission. Permission for the publication or other use of these materials may be granted only by the Licensors and copyright holders.

Cover design by John Lasseter, Lucasfilm, Ltd.

4.2 BSD edition:

First Printing	July 1984
Second Printing	December 1984
Third Printing	September 1985
Fourth Printing	March 1986

4.3 BSD edition:

First Printing	November 1986
Second Printing	June 1988

## UNIX Documents

URM	User Reference Manual	PS1:6	Berkeley Software Architecture Manual (4.3 Edition)
	man section 1 (commands)	PS1:7	Introductory 4.3BSD Interprocess Communication
	man section 6 (games)	PS1:8	Advanced 4.3BSD Interprocess Communication
	man section 7 (miscellaneous)	PS1:9	Lint, A C Program Checker
USD	User Supplementary Documents	PS1:10	ADB Tutorial
USD:1	Unix for Beginners	PS1:11	Debugging with dbx
USD:2	Learn - Computer-Aided Instruction	PS1:12	Make
USD:3	Introduction to the UNIX Shell	PS1:13	Revision Control System (RCS)
USD:4	Introduction to the C shell	PS1:14	Source Code Control System (SCCS)
USD:5	DC - Interactive Desk Calculator	PS1:15	YACC: Yet Another Compiler-Compiler
USD:6	BC - Arbitrary Precision Desk-Calculator	PS1:16	LEX - A Lexical Analyzer Generator
USD:7	Mail Reference Manual	PS1:17	M4 Macro Processor
USD:8	MH Message Handling System	PS1:18	curses library
USD:9	How to Read the Network News		
USD:10	How to Use USENET Effectively	PS2	Programmer Supplementary Documents, part 2
USD:11	Notesfile Reference Manual	PS2:1	The Unix Time-Sharing System
USD:12	Tutorial Introduction to "ed"	PS2:2	UNIX 32/V - Summary
USD:13	Advanced Editing on Unix	PS2:3	Unix Programming - Second Edition
USD:14	Edit: A Tutorial	PS2:4	Unix Implementation
USD:15	Introduction to Display Editing with Vi	PS2:5	The Unix I/O System
USD:16	Ex Reference Manual (Version 3.7)	PS2:6	Programming Language EFL
USD:17	Jove Manual for UNIX Users	PS2:7	Berkeley FP User's Manual
USD:18	SED - A Non-interactive Text Editor	PS2:8	Ratfor - Preprocessor for Rational FORTRAN
USD:19	AWK - Pattern Scanning/Processing Language	PS2:9	The FRANZ LISP Manual
USD:20	Using the -ms Macros with Troff and Nroff	PS2:10	Ingres (Version 8) Reference Manual
USD:21	Revised Version of -ms		
USD:22	Writing Papers with nroff using -me	SMM	System Manager's Manual
USD:23	-me Reference Manual		man section 8 (system administration)
USD:24	NROFF/TROFF User's Manual	SMM:1	Installing and Operating 4.3BSD
USD:25	TROFF Tutorial	SMM:2	Building 4.3BSD Systems with Config
USD:26	Typesetting Mathematics (eqn)	SMM:3	Using ADB to Debug the Kernel
USD:27	Typesetting Mathematics - User's Guide	SMM:4	Disc Quotas
USD:28	Tbl - A Program to Format Tables	SMM:5	Fsck - File System Check Program
USD:29	Refer - A Bibliography System	SMM:6	Line Printer Spooler Manual
USD:30	Some Applications of Inverted Indexes ...	SMM:7	Sendmail Installation and Operation
USD:31	BIB - Bibliography Formatting Program	SMM:8	Timed Installation and Operation
USD:32	Writing Tools - STYLE and DICTION	SMM:9	UUCP Implementation Description
USD:33	A Guide to the Dungeons of Doom	SMM:10	USENET Version B Installation
USD:34	Star Trek	SMM:11	Name Server Operations Guide
		SMM:12	Bug Fixes and Changes in 4.3BSD
PRM	Programmer Reference Manual	SMM:13	Changes to the Kernel in 4.3BSD
	man sections 2 (system calls)	SMM:14	A Fast File System for UNIX
	man sections 3 (library routines)	SMM:15	4.3BSD Networking Implementation Notes
	man sections 4 (devices, special files)	SMM:16	Sendmail - An Internetwork Mail Router
	man sections 5 (file formats)	SMM:17	On the Security of UNIX
PS1	Programmer Supplementary Docs, part 1	SMM:18	Password Security - A Case History
PS1:1	C Language - Reference Manual	SMM:19	A Tour Through the Portable C Compiler
PS1:2	Fortran 77	SMM:20	Writing NROFF Terminal Descriptions
PS1:3	f77 I/O Library	SMM:21	A Dial-Up Network of UNIX Systems
PS1:4	Berkeley Pascal User's Manual	SMM:22	Berkeley Time Synchronization Protocol
PS1:5	Vax Assembler Reference Manual		





# **UNIX System Manager's Manual (SMM)**

## **4.3 Berkeley Software Distribution Virtual VAX-11 Version**

**April, 1986**

**Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California  
Berkeley, California 94720**

Copyright 1979, 1980, 1983, 1986 Regents of the University of California. Permission to copy these documents or any portion thereof as necessary for licensed use of the software is granted to licensees of this software, provided this copyright notice and statement of permission are included.

Documents SMM:17, 18, and 21 are copyright 1979, AT&T Bell Laboratories, Incorporated. Documents SMM:9 and 19 are modifications of earlier documents that are copyrighted 1979 by AT&T Bell Laboratories, Incorporated. Holders of UNIX<sup>TM</sup>/32V, System III, or System V software licenses are permitted to copy these documents, or any portion of them, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

Document SMM:10 is part of the user contributed software.

This manual reflects system enhancements made at Berkeley and sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in these documents are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

## UNIX System Manager's Manual (SMM)

### 4.3 Berkeley Software Distribution, Virtual VAX-11 Version

April, 1986

This volume contains manual pages and supplementary documents useful to system administrators. The information in these documents applies to the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

(8)

#### Reference Manual – Section 8

Section 8 of the UNIX Programmer's Manual contains information related to system operation, administration, and maintenance.

#### System Installation and Administration

##### Installing and Operating 4.3BSD on the VAX SMM:1

The definitive reference document for those occasions when you find you need to start over again.

##### Building 4.3BSD UNIX Systems with *Config* SMM:2

In-depth discussions of the use and operation of the *config* program, and how to build your very own Unix kernel.

##### Using ADB to Debug the Kernel SMM:3

Techniques for figuring out after the fact why the kernel crashed.

##### Disc Quotas in a UNIX Environment SMM:4

A light introduction to the techniques for limiting the use of disc resources.

##### Fsck – The UNIX File System Check Program SMM:5

A reference document for using the *fsck* program during times of file system distress.

##### Line Printer Spooler Manual SMM:6

This document describes the structure and installation procedure for the line printer spooling system.

##### Sendmail Installation and Operation Guide SMM:7

The last word in installing and operating the *sendmail* program.

##### Timed Installation and Operation Guide SMM:8

Describes how to maintain time synchronization between machines in a local network.

## SMM Contents

UUCP Implementation Description	SMM:9
Describes the implementation of uucp; for the installer and administrator.	
USENET Version B Installation	SMM:10
How to install and maintain the News system.	
Name Server Operations Guide	SMM:11
If you have a network this will be of interest.	
<b>Supporting Documentation</b>	
Bug Fixes and Changes in 4.3BSD	SMM:12
This document summarizes changes visible to the user accustomed to 4.2BSD.	
Changes to the Kernel in 4.3BSD	SMM:13
A summary for the hard-core of changes in the kernel from 4.2BSD to 4.3BSD.	
A Fast File System for UNIX	SMM:14
A description of the 4.2BSD file system organization, design and implementation.	
4.3BSD Networking Implementation Notes	SMM:15
A concise description of the system interfaces used within the networking subsystem.	
Sendmail – An Internetwork Mail Router	SMM:16
An overview document on the design and implementation of <i>sendmail</i> .	
On the Security of UNIX	SMM:17
Hints on how to break UNIX, and how to avoid your system being broken.	
Password Security – A Case History	SMM:18
How the bad guys used to be able to break the password algorithm, and why they cannot now (at least not so easily).	
A Tour Through the Portable C Compiler	SMM:19
How the portable C compiler works inside.	
Writing NROFF Terminal Descriptions	SMM:20
A description of how to add a printer with new characteristics to Version 7 <i>nroff</i> .	
A Dial-Up Network of UNIX Systems	SMM:21
Describes UUCP, a program for communicating files between UNIX systems.	
The Berkeley UNIX Time Synchronization Protocol	SMM:22
The protocols and algorithms used by <i>timed</i> , the network time synchronization daemon.	

## TABLE OF CONTENTS

### 8. System Maintenance

intro	introduction to system maintenance and operation commands
ac	login accounting
adduser	procedure for adding new users
arff	archiver and copier for floppy
arp	address resolution display and control
bad144	read/write dec standard 144 bad sector information
badsect	create files to contain bad sectors
bugfiler	file bug reports in folders automatically
catman	create the cat files for the manual
chown	change owner
clri	clear i-node
comsat	biff server
config	build system configuration files
crash	what happens when the system crashes
cron	clock daemon
dcheck	file system directory consistency check
diskpart	calculate default disk partition sizes
dmesg	collect system diagnostic messages to form error log
drtest	standalone disk test program
dump	incremental file system dump
dumpfs	dump file system information
edquota	edit user quotas
fastboot	reboot/halt the system without checking the disks
fingerd	remote user information server
format	how to format disk packs
fsck	file system consistency check and interactive repair
ftpd	DARPA Internet File Transfer Protocol server
gettable	get NIC format host tables from a host
getty	set terminal mode
halt	stop the processor
htable	convert NIC standard format host tables
ichck	file system storage consistency check
ifconfig	configure network interface parameters
implog	IMP log interpreter
implogd	IMP logger process
inetd	internet "super-server"
init	process control initialization
kgmon	generate a dump of the operating system's profile buffers
lpc	line printer control program
lpd	line printer daemon
makedev	make system special files
makekey	generate encryption key
mkfs	construct a file system
mkhosts	generate hashed host table
mklost+found	make a lost+found directory for fsck
mknod	build special file
mkpasswd	generate hashed password table
mkproto	construct a prototype file system
mount	mount and dismount file system
named	Internet domain name server
ncheck	generate names from i-numbers
newfs	construct a new file system
pac	printer/plotter accounting information



ping	send ICMP ECHO_REQUEST packets to network hosts
pstat	print system facts
quot	summarize file system ownership
quotacheck	file system quota consistency checker
quotaon	turn file system quotas on and off
rc	command script for auto-reboot and daemons
rdump	file system dump across the network
reboot	UNIX bootstrapping procedures
renice	alter priority of running processes
repquota	summarize quotas for a file system
restore	incremental file system restore
rexecd	remote execution server
rlogind	remote login server
rmt	remote magtape protocol module
route	manually manipulate the routing tables
routed	network routing daemon
rrestore	restore a file system dump across the network
rshd	remote shell server
rwhod	system status server
rxformat	format floppy disks
sa	system accounting
savecore	save a core dump of the operating system
sendmail	send mail over the internet
shutdown	close down the system at a given time
slattach	attach serial lines as network interfaces
sticky	persistent text and append-only directories
swapon	specify additional device for paging and swapping
sync	update the super block
syslogd	log systems messages
talkd	remote user communication server
telnetd	DARPA TELNET protocol server
tftpd	DARPA Trivial File Transfer Protocol server
timed	time server daemon
timedc	timed control program
trpt	transliterate protocol trace
trsp	transliterate sequenced packet protocol trace
tunefs	tune up an existing file system
update	periodically update the super block
uucico	transfer files queued by uucp or uux
uuclean	uucp spool directory clean-up
uupoll	poll a remote UUCP site
uusnap	show snapshot of the UUCP system
uuxqt	UUCP execution file interpreter
vipw	edit the password file
XNSrouted	NS Routing Information Protocol daemon

**NAME**

intro – introduction to system maintenance and operation commands

**DESCRIPTION**

This section contains information related to system operation and maintenance. It describes commands used to create new file systems, *newfs*, verify the integrity of the file systems, *fsck*, control disk usage, *edquota*, maintain system backups, *dump*, and recover files when disks die an untimely death, *restore*. The section *format* should be consulted when formatting disk packs. Network related services are distinguished as 8C. The section *crash* should be consulted in understanding how to interpret system crash dumps.

**NAME**

ac - login accounting

**SYNOPSIS**

*/etc/ac* [ -w *wtmp* ] [ -p ] [ -d ] [ *people* ] ...

**DESCRIPTION**

*Ac* produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. -w is used to specify an alternate *wtmp* file. -p prints individual totals; without this option, only totals are printed. -d causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

The accounting file */usr/adm/wtmp* is maintained by *init* and *login*. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

**FILES**

*/usr/adm/wtmp*

**SEE ALSO**

*init*(8), *sa*(8), *login*(1), *utmp*(5).

**NAME**

adduser – procedure for adding new users

**DESCRIPTION**

A new user must choose a login name, which must not already appear in */etc/passwd*. An account can be added by editing a line into the *passwd* file; this must be done with the *passwd* file locked e.g. by using *vipw*(8).

A new user is given a group and user id. User id's should be distinct across a system, since they are used to control access to files. Typically, users working on similar projects will be put in the same group. Thus at UCB we have groups for system staff, faculty, graduate students, and a few special groups for large projects. System staff is group "10" for historical reasons, and the super-user is in this group.

A skeletal account for a new user "ernie" would look like:

```
ernie::235:20:& Kovacs,508E,7925,6428202:/mnt/grad/ernie:/bin/csh
```

The first field is the login name "ernie". The next field is the encrypted password which is not given and must be initialized using *passwd*(1). The next two fields are the user and group id's. Traditionally, users in group 20 are graduate students and have account names with numbers in the 200's. The next field gives information about ernie's real name, office and office phone and home phone. This information is used by the *finger*(1) program. From this information we can tell that ernie's real name is "Ernie Kovacs" (the & here serves to repeat "ernie" with appropriate capitalization), that his office is 508 Evans Hall, his extension is x2-7925, and this his home phone number is 642-8202. You can modify the *finger*(1) program if necessary to allow different information to be encoded in this field. The UCB version of *finger* knows several things particular to Berkeley – that phone extensions start "2-", that offices ending in "E" are in Evans Hall and that offices ending in "C" are in Cory Hall. The *chfn*(1) program allows users to change this information.

The final two fields give a login directory and a login shell name. Traditionally, user files live on a file system different from */usr*. Typically the user file systems are mounted on a directories in the root named sequentially starting from the beginning of the alphabet, eg */a*, */b*, */c*, etc. On each such file system there are subdirectories there for each group of users, i.e.: */a/staff* and */b/prof*. This is not strictly necessary but keeps the number of files in the top level directories reasonably small.

The login shell will default to */bin/sh* if none is given. Most users at Berkeley choose */bin/csh* so this is usually specified here. The *chsh*(1) program allows users to change their login shell to one of the shells in the approved list given in */etc/shells*.

It is useful to give new users some help in getting started, supplying them with a few skeletal files such as *.profile* if they use */bin/sh*, or *.cshrc* and *.login* if they use */bin/csh*. The directory */usr/skel* contains skeletal definitions of such files. New users should be given copies of these files which, for instance, arrange to use *tset*(1) automatically at each login.

**FILES**

<i>/etc/passwd</i>	password file
<i>/usr/skel</i>	skeletal login directory

**SEE ALSO**

*passwd*(1), *finger*(1), *chsh*(1), *chfn*(1), *passwd*(5), *vipw*(8)

**BUGS**

User information should be stored in its own data base separate from the password file.

**NAME**

*arff*, *flcopy* – archiver and copier for floppy

**SYNOPSIS**

```
/etc/arff [ key ] [ name ... ]
/etc/flcopy [ -h ] [ -tn ]
```

**DESCRIPTION**

*Arff* saves and restores files on VAX console media (the console floppy on the VAX 11/780 and 785, the cassette on the 11/730, and the console RL02 on the 8600/8650). Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file names specifying which files are to be dumped or restored. The default options are correct for the RX01 floppy on the 780; for other console media, the *f* and *m* flags are required.

Files names have restrictions, because of radix50 considerations. They must be in the form 1-6 alphanumerics followed by "." followed by 0-3 alphanumerics. Case distinctions are lost. Only the trailing component of a pathname is used.

The function portion of the key is specified by one of the following letters:

- r** The named files are replaced where found on the floppy, or added taking up the minimal possible portion of the first empty spot on the floppy.
- x** The named files are extracted from the floppy.
- d** The named files are deleted from the floppy. *Arff* will combine contiguous deleted files into one empty entry in the rt-11 directory.
- t** The names of the specified files are listed each time they occur on the floppy. If no file argument is given, all of the names on the floppy are listed.

The following characters may be used in addition to the letter which selects the function desired.

- v** The *v* (verbose) option, when used with the *t* function gives more information about the floppy entries than just the name.
- f** causes *arff* to use the next argument as the name of the archive instead of */dev/floppy*.
- m** causes *arff* not to use the mapping algorithm employed in interleaving sectors around a floppy disk. In conjunction with the *f* option it may be used for extracting files from rt11 formatted cartridge disks, for example. It may also be used to speed up reading from and writing to rx02 floppy disks, by using the 'c' device instead of the 'b' device. It must be used with TU58 or RL02 media.
- c** causes *arff* to create a new directory on the floppy, effectively deleting all previously existing files.

*Flcopy* copies the console floppy disk (opened as */dev/floppy*) to a file created in the current directory, named "floppy", then prints the message "Change Floppy, hit return when done". Then *flcopy* copies the local file back out to the floppy disk.

The *-h* option to *flcopy* causes it to open a file named "floppy" in the current directory and copy it to */dev/floppy*; the *-t* option causes only the first *n* tracks to participate in a copy.

**FILES**

*/dev/floppy* or */dev/rxx??*  
*floppy* (in current directory)



**SEE ALSO**

crl(4), fl(4), rx(4), tu(4), rxformat(8V)

**AUTHORS**

Keith Sklower, Richard Tuck

**BUGS**

Device errors are handled ungracefully.

**NAME**

**arp** – address resolution display and control

**SYNOPSIS**

```
arp hostname
arp -a [ vmunix ] [ kmem ]
arp -d hostname
arp -s hostname ether_addr [ temp ] [ pub ] [ trail ]
arp -f filename
```

**DESCRIPTION**

The *arp* program displays and modifies the Internet-to-Ethernet address translation tables used by the address resolution protocol (*arp*(4p)).

With no flags, the program displays the current ARP entry for *hostname*. The host may be specified by name or by number, using Internet dot notation. With the **-a** flag, the program displays all of the current ARP entries by reading the table from the file *kmem* (default /dev/kmem) based on the kernel file *vmunix* (default /vmunix).

With the **-d** flag, a super-user may delete an entry for the host called *hostname*.

The **-s** flag is given to create an ARP entry for the host called *hostname* with the Ethernet address *ether\_addr*. The Ethernet address is given as six hex bytes separated by colons. The entry will be permanent unless the word *temp* is given in the command. If the word *pub* is given, the entry will be "published"; i.e., this system will act as an ARP server, responding to requests for *hostname* even though the host address is not its own. The word *trail* indicates that trailer encapsulations may be sent to this host.

The **-f** flag causes the file *filename* to be read and multiple entries to be set in the ARP tables. Entries in the file should be of the form

```
hostname ether_addr [ temp ] [ pub ] [ trail ]
```

with argument meanings as given above.

**SEE ALSO**

inet(3N), arp(4P), ifconfig(8C)

## NAME

**bad144** - read/write dec standard 144 bad sector information

## SYNOPSIS

```
/etc/bad144 [ -f ] [ -c ] [ -v ] disktype disk [ sno [ bad ... ] ]
/etc/bad144 -a [ -f ] [ -c ] [ -v ] disktype disk [ bad ... ]
```

## DESCRIPTION

*Bad144* can be used to inspect the information stored on a disk that is used by the disk drivers to implement bad sector forwarding. The format of the information is specified by DEC standard 144, as follows.

The bad sector information is located in the first 5 even numbered sectors of the last track of the disk pack. There are five identical copies of the information, described by the *dkbad* structure.

Replacement sectors are allocated starting with the first sector before the bad sector information and working backwards towards the beginning of the disk. A maximum of 126 bad sectors are supported. The position of the bad sector in the bad sector table determines the replacement sector to which it corresponds. The bad sectors must be listed in ascending order.

The bad sector information and replacement sectors are conventionally only accessible through the "c" file system partition of the disk. If that partition is used for a file system, the user is responsible for making sure that it does not overlap the bad sector information or any replacement sectors. Thus, one track plus 126 sectors must be reserved to allow use of all of the possible bad sector replacements.

The bad sector structure is as follows:

```
struct dkbad (
    long      bt_csn;                /* cartridge serial number */
    u_short   bt_mbz;                /* unused; should be 0 */
    u_short   bt_flag;               /* -1 => alignment cartridge */
    struct bt_bad {
        u_short bt_cyl;              /* cylinder number of bad sector */
        u_short bt_trksec;           /* track and sector number */
    } bt_bad[126];
);
```

Unused slots in the *bt\_bad* array are filled with all bits set, a putatively illegal value.

*Bad144* is invoked by giving a device type (e.g. rk07, rm03, rm05, etc.), and a device name (e.g. hk0, hp1, etc.). With no optional arguments it reads the first sector of the last track of the corresponding disk and prints out the bad sector information. It issues a warning if the bad sectors are out of order. *Bad144* may also be invoked with a serial number for the pack and a list of bad sectors. It will write the supplied information into all copies of the bad-sector file, replacing any previous information. Note, however, that *bad144* does not arrange for the specified sectors to be marked bad in this case. This procedure should only be used to restore known bad sector information which was destroyed. It is necessary to reboot before any change will take effect.

With the *-a* option, the argument list consists of new bad sectors to be added to an existing list. The new sectors are sorted into the list, which must have been in order. Replacement sectors are moved to accommodate the additions; the new replacement sectors are cleared. The entire process is described as it happens in gory detail if *-v* (verbose) is given. The *-c* option forces an attempt to copy the old sector to the replacement, and may be useful when replacing an unreliable sector.

If the disk is an RP06, RM03, RM05, Fujitsu Eagle, or SMD disk on a Massbus, the `-f` option may be used to mark the new bad sectors as "bad" by reformatting them as unusable sectors. **NOTE:** this can be done safely only when there is no other disk activity, preferably while running single-user. This option is required unless the sectors have already been marked bad, or the system will not be notified that it should use the replacement sector.

**SEE ALSO**

`badsect(8)`, `format(8V)`

**BUGS**

It should be possible to format disks on-line under UNIX.

It should be possible to mark bad sectors on drives of all type.

On an 11/750, the standard bootstrap drivers used to boot the system do not understand bad sectors, handle ECC errors, or the special SSE (skip sector) errors of RM80-type disks. This means that none of these errors can occur when reading the file `/vmunix` to boot. Sectors 0-15 of the disk drive must also not have any of these errors.

The drivers which write a system core image on disk after a crash do not handle errors; thus the crash dump area must be free of errors and bad sectors.

**NAME**

badsect – create files to contain bad sectors

**SYNOPSIS**

/etc/badsect bmdir sector ...

**DESCRIPTION**

*Badsect* makes a file to contain a bad sector. Normally, bad sectors are made inaccessible by the standard formatter, which provides a forwarding table for bad sectors to the driver; see *bad144(8)* for details. If a driver supports the bad blocking standard it is much preferable to use that method to isolate bad blocks, since the bad block forwarding makes the pack appear perfect, and such packs can then be copied with *dd(1)*. The technique used by this program is also less general than bad block forwarding, as *badsect* can't make amends for bad blocks in the i-list of file systems or in swap areas.

On some disks, adding a sector which is suddenly bad to the bad sector table currently requires the running of the standard DEC formatter. Thus to deal with a newly bad block or on disks where the drivers do not support the bad-blocking standard *badsect* may be used to good effect.

*Badsect* is used on a quiet file system in the following way: First mount the file system, and change to its root directory. Make a directory BAD there. Run *badsect* giving as argument the BAD directory followed by all the bad sectors you wish to add. (The sector numbers must be relative to the beginning of the file system, but this is not hard as the system reports relative sector numbers in its console error messages.) Then change back to the root directory, unmount the file system and run *fsck(8)* on the file system. The bad sectors should show up in two files or in the bad sector files and the free list. Have *fsck* remove files containing the offending bad sectors, but do not have it remove the BAD/nnnnn files. This will leave the bad sectors in only the BAD files.

*Badsect* works by giving the specified sector numbers in a *mknod(2)* system call, creating an illegal file whose first block address is the block containing bad sector and whose name is the bad sector number. When it is discovered by *fsck* it will ask "HOLD BAD BLOCK"? A positive response will cause *fsck* to convert the inode to a regular file containing the bad block.

**SEE ALSO**

bad144(8), fsck(8), format(8V)

**DIAGNOSTICS**

*Badsect* refuses to attach a block that resides in a critical area or is out of range of the file system. A warning is issued if the block is already in use.

**BUGS**

If more than one sector which comprise a file system fragment are bad, you should specify only one of them to *badsect*, as the blocks in the bad sector files actually cover all the sectors in a file system fragment.



**NAME**

bugfiler – file bug reports in folders automatically

**SYNOPSIS**

bugfiler [ mail directory ]

**DESCRIPTION**

*Bugfiler* is a program to automatically intercept bug reports, summarize them and store them in the appropriate sub directories of the mail directory specified on the command line or the (system dependent) default. It is designed to be compatible with the Rand MH mail system. *Bugfiler* is normally invoked by the mail delivery program through *aliases*(5) with a line such as the following in */usr/lib/aliases*.

```
bugs: "|bugfiler /usr/bugs/mail"
```

It reads the message from the standard input or the named file, checks the format and returns mail acknowledging receipt or a message indicating the proper format. Valid reports are then summarized and filed in the appropriate folder; improperly formatted messages are filed in a folder named "errors." Program maintainers can then log onto the system and check the summary file for bugs that pertain to them. Bug reports should be submitted in RFC822 format and are must contain the following header lines to be properly indexed:

```
Date: <date the report is received>
From: <valid return address>
Subject: <short summary of the problem>
Index: <source directory>/<source file> <version> [Fix]
```

In addition, the body of the message must contain a line which begins with "Description:" followed by zero or more lines describing the problem in detail and a line beginning with "Repeat-By:" followed by zero or more lines describing how to repeat the problem. If the keyword 'Fix' is specified in the 'Index' line, then there must also be a line beginning with "Fix:" followed by a diff of the old and new source files or a description of what was done to fix the problem.

The 'Index' line is the key to the filing mechanism. The source directory name must match one of the folder names in the mail directory. The message is then filed in this folder and a line appended to the summary file in the following format:

```
<folder name>/<message number>    <Index info>
                                     <Subject info>
```

The bug report may also be redistributed according to the index. If the file *maildir/.redist* exists, it is examined for a line beginning with the index name followed with a tab. The remainder of this line contains a comma-separated list of mail addresses which should receive copies of bugs with this index. The list may be continued onto multiple lines by ending each but the last with a backslash ('\').

**FILES**

<i>/usr/lib/sendmail</i>	mail delivery program
<i>/usr/lib/unixtomh</i>	converts unix mail format to mh format
<i>maildir/.ack</i>	the message sent in acknowledgement
<i>maildir/.format</i>	the message sent when format errors are detected
<i>maildir/.redist</i>	the redistribution list
<i>maildir/summary</i>	the summary file
<i>maildir/Bf?????</i>	temporary copy of the input message
<i>maildir/Rp?????</i>	temporary file for the reply message.

**SEE ALSO**

*mh*(1), *newaliases*(1), *aliases*(5)

**BUGS**

Since mail can be forwarded in a number of different ways, *bugfiler* does not recognize forwarded mail and will reply/complain to the forwarder instead of the original sender unless there is a 'Reply-To' field in the header.

Duplicate messages should be discarded or recognized and put somewhere else.

## NAME

catman - create the cat files for the manual

## SYNOPSIS

```
/etc/catman [ -p ] [ -n ] [ -w ] [ -M path ] [ sections ]
```

## DESCRIPTION

*Catman* creates the preformatted versions of the on-line manual from the nroff input files. Each manual page is examined and those whose preformatted versions are missing or out of date are recreated. If any changes are made, *catman* will recreate the *whatis* database.

If there is one parameter not starting with a '-', it is taken to be a list of manual sections to look in. For example

```
catman 123
```

will cause the updating to only happen to manual sections 1, 2, and 3.

## Options:

- n prevents creations of the *whatis* database.
- p prints what would be done instead of doing it.
- w causes only the *whatis* database to be created. No manual reformatting is done.
- M updates manual pages located in the set of directories specified by *path* (*/usr/man* by default). *Path* has the form of a colon (':') separated list of directory names, for example '*/usr/local/man:/usr/man*'. If the environment variable '*MANPATH*' is set, its value is used for the default path.

If the nroff source file contains only a line of the form '.so manx/yyy.x', a symbolic link is made in the catx directory to the appropriate preformatted manual page. This feature allows easy distribution of the preformatted manual pages among a group of associated machines with *rdist(1)*. The nroff sources need not be distributed to all machines, thus saving the associated disk space. As an example, consider a local network with 5 machines, called mach1 through mach5. Suppose mach3 has the manual page nroff sources. Every night, mach3 runs *catman* via *cron(8)* and later runs *rdist* with a distfile that looks like:

```
MANSLAVES = ( mach1 mach2 mach4 mach5 )
```

```
MANUALS = (/usr/man/cat[1-8no] /usr/man/whatis)
```

```
$(MANUALS) -> $(MANSLAVES)
install -R;
notify root;
```

## FILES

<i>/usr/man</i>	default manual directory location
<i>/usr/man/man?/*.*</i>	raw (nroff input) manual sections
<i>/usr/man/cat?/*.*</i>	preformatted manual pages
<i>/usr/man/whatis</i>	<i>whatis</i> database
<i>/usr/lib/makewhatis</i>	command script to make <i>whatis</i> database

## SEE ALSO

*man(1)*, *cron(8)*, *rdist(1)*

## BUGS

Acts oddly on nights with full moons.

**NAME**

chown - change owner

**SYNOPSIS**

/etc/chown [ -f -R ] owner[.group] file ...

**DESCRIPTION**

*Chown* changes the owner of the *files* to *owner*. The owner may be either a decimal UID or a login name found in the password file. An optional group may also be specified. The group may be either a decimal GID or a group name found in the group-ID file.

Only the super-user can change owner, in order to simplify accounting procedures. No errors are reported when the -f (force) option is given.

When the -R option is given, *chown* recursively descends its directory arguments setting the specified owner. When symbolic links are encountered, their ownership is changed, but they are not traversed.

**FILES**

/etc/passwd

**SEE ALSO**

chgrp(1), chown(2), passwd(5), group(5)

**NAME**

`clri` - clear i-node

**SYNOPSIS**

`/etc/clri filesystem i-number ...`

**DESCRIPTION**

**N.B.:** *Clri* is obsoleted for normal file system repair work by *fsck*(8).

*Clri* writes zeros on the i-nodes with the decimal *i-numbers* on the *filesystem*. After *clri*, any blocks in the affected file will show up as 'missing' in an *icheck*(8) of the *filesystem*.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

**SEE ALSO**

*icheck*(8)

**BUGS**

If the file is open, *clri* is likely to be ineffective.



**NAME**

comsat - biff server

**SYNOPSIS**

/etc/comsat

**DESCRIPTION**

*Comsat* is the server process which receives reports of incoming mail and notifies users if they have requested this service. *Comsat* receives messages on a datagram port associated with the "biff" service specification (see *services*(5) and *inetd*(8)). The one line messages are of the form

user@mailbox-offset

If the *user* specified is logged in to the system and the associated terminal has the owner execute bit turned on (by a "biff y"), the *offset* is used as a seek offset into the appropriate mailbox file and the first 7 lines or 560 characters of the message are printed on the user's terminal. Lines which appear to be part of the message header other than the "From", "To", "Date", or "Subject" lines are not included in the displayed message.

**FILES**

/etc/utmp      to find out who's logged on and on what terminals

**SEE ALSO**

biff(1), inetd(8)

**BUGS**

The message header filtering is prone to error. The density of the information presented is near the theoretical minimum.

Users should be notified of mail which arrives on other machines than the one to which they are currently logged in.

The notification should appear in a separate window so it does not mess up the screen.

**NAME**

`config` - build system configuration files

**SYNOPSIS**

`/etc/config [ -p ] SYSTEM_NAME`

**DESCRIPTION**

*Config* builds a set of system configuration files from a short file which describes the sort of system that is being configured. It also takes as input a file which tells *config* what files are needed to generate a system. This can be augmented by a configuration specific set of files that give alternate files for a specific machine. (see the FILES section below) If the `-p` option is supplied, *config* will configure a system for profiling; c.f. *kgmon*(8) and *gprof*(1).

*Config* should be run from the `conf` subdirectory of the system source (usually `/sys/conf`). Its argument is the name of a system configuration file containing device specifications, configuration options and other system parameters for one system configuration. *Config* assumes that there is already a directory `..SYSTEM_NAME` created and it places all its output files in there. The output of *config* consists of a number of files; for the VAX, they are: *ioconf.c* contains a description of what I/O devices are attached to the system; *ubglue.s* contains a set of interrupt service routines for devices attached to the UNIBUS; *ubvec.s* contains offsets into a structure used for counting per-device interrupts; *Makefile* is a file used by *make*(1) in building the system; a set of header files contain definitions of the number of various devices that will be compiled into the system; and a set of swap configuration files contain definitions for the disk areas to be used for swapping, the root file system, argument processing, and system dumps.

After running *config*, it is necessary to run "make depend" in the directory where the new makefile was created. *Config* prints a reminder of this when it completes.

If any other error messages are produced by *config*, the problems in the configuration file should be corrected and *config* should be run again. Attempts to compile a system that had configuration errors are likely to meet with failure.

**FILES**

`/sys/conf/Makefile.vax` generic makefile for the VAX  
`/sys/conf/files` list of common files system is built from  
`/sys/conf/files.vax` list of VAX specific files  
`/sys/conf/devices.vax` name to major device mapping file for the VAX  
`/sys/conf/files.ERNIE` list of files specific to ERNIE system

**SEE ALSO**

"Building 4.3BSD UNIX System with Config"  
 The SYNOPSIS portion of each device in section 4.

**BUGS**

The line numbers reported in error messages are usually off by one.

**NAME**

crash – what happens when the system crashes

**DESCRIPTION**

This section explains what happens when the system crashes and (very briefly) how to analyze crash dumps.

When the system crashes voluntarily it prints a message of the form

panic: why i gave up the ghost

on the console, takes a dump on a mass storage peripheral, and then invokes an automatic reboot procedure as described in *reboot*(8). (If auto-reboot is disabled on the front panel of the machine the system will simply halt at this point.) Unless some unexpected inconsistency is encountered in the state of the file systems due to hardware or software failure, the system will then resume multi-user operations.

The system has a large number of internal consistency checks; if one of these fails, then it will panic with a very short message indicating which one failed. In many instances, this will be the name of the routine which detected the error, or a two-word description of the inconsistency. A full understanding of most panic messages requires perusal of the source code for the system.

The most common cause of system failures is hardware failure, which can reflect itself in different ways. Here are the messages which are most likely, with some hints as to causes. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

**ilnit** This cryptic panic message results from a failure to mount the root filesystem during the bootstrap process. Either the root filesystem has been corrupted, or the system is attempting to use the wrong device as root filesystem. Usually, an alternate copy of the system binary or an alternate root filesystem can be used to bring up the system to investigate.

**Can't exec /etc/init**

This is not a panic message, as reboots are likely to be futile. Late in the bootstrap procedure, the system was unable to locate and execute the initialization process, *init*(8). The root filesystem is incorrect or has been corrupted, or the mode or type of */etc/init* forbids execution.

**IO err in push****hard IO err in swap**

The system encountered an error trying to write to the paging device or an error in reading critical information from a disk drive. The offending disk should be fixed if it is broken or unreliable.

**realloccg: bad optim****ialloc: dup alloc****allocgblk: cyl groups corrupted****iallocg: map corrupted****free: freeing free block****free: freeing free frag****ifree: freeing free inode****allocg: map corrupted**

These panic messages are among those that may be produced when filesystem inconsistencies are detected. The problem generally results from a failure to repair damaged filesystems after a crash, hardware failures, or other condition that should not normally occur. A filesystem check will normally correct the problem.

**timeout table overflow**

This really shouldn't be a panic, but until the data structure involved is made to be extensible, running out of entries causes a crash. If this happens, make the timeout table bigger.

**KSP not valid****SBI fault****CHM? in kernel**

These indicate either a serious bug in the system or, more often, a glitch or failing hardware. If SBI faults recur, check out the hardware or call field service. If the other faults recur, there is likely a bug somewhere in the system, although these can be caused by a flakey processor. Run processor microdiagnostics.

**machine check %x:**  
*description*
*machine dependent machine-check information*

Machine checks are different on each type of CPU. Most of the internal processor registers are saved at the time of the fault and are printed on the console. For most processors, there is one line that summarizes the type of machine check. Often, the nature of the problem is apparent from this message and/or the contents of key registers. The VAX Hardware Handbook should be consulted, and, if necessary, your friendly field service people should be informed of the problem.

**trap type %d, code=%x, pc=%x**

A unexpected trap has occurred within the system; the trap types are:

0	reserved addressing fault
1	privileged instruction fault
2	reserved operand fault
3	bpt instruction fault
4	xfc instruction fault
5	system call trap
6	arithmetic trap
7	ast delivery trap
8	segmentation fault
9	protection fault
10	trace trap
11	compatibility mode fault
12	page fault
13	page table fault

The favorite trap types in system crashes are trap types 8 and 9, indicating a wild reference. The code is the referenced address, and the pc at the time of the fault is printed. These problems tend to be easy to track down if they are kernel bugs since the processor stops cold, but random flakiness seems to cause this sometimes. The debugger can be used to locate the instruction and subroutine corresponding to the PC value. If that is insufficient to suggest the nature of the problem, more detailed examination of the system status at the time of the trap usually can produce an explanation.

**init died**

The system initialization process has exited. This is bad news, as no new users will then be able to log in. Rebooting is the only fix, so the system just does it right away.

**out of mbufs: map full**

The network has exhausted its private page map for network buffers. This usually

indicates that buffers are being lost, and rather than allow the system to slowly degrade, it reboots immediately. The map may be made larger if necessary.

That completes the list of panic types you are likely to see.

When the system crashes it writes (or at least attempts to write) an image of memory into the back end of the dump device, usually the same as the primary swap area. After the system is rebooted, the program *savecore(8)* runs and preserves a copy of this core image and the current system in a specified directory for later perusal. See *savecore(8)* for details.

To analyze a dump you should begin by running *adb(1)* with the *-k* flag on the system load image and core dump. If the core image is the result of a panic, the panic message is printed. Normally the command “*\$c*” will provide a stack trace from the point of the crash and this will provide a clue as to what went wrong. A more complete discussion of system debugging is impossible here. See, however, “Using ADB to Debug the UNIX Kernel”.

#### SEE ALSO

*adb(1)*, *reboot(8)*

*VAX 11/780 System Maintenance Guide* and *VAX Hardware Handbook* for more information about machine checks.

*Using ADB to Debug the UNIX Kernel*

**NAME**

cron - clock daemon

**SYNOPSIS**

/etc/cron

**DESCRIPTION**

*Cron* executes commands at specified dates and times according to the instructions in the files /usr/lib/crontab and /usr/lib/crontab.local. None, either one, or both of these files may be present. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file /etc/rc; see *init*(8).

The crontab files consist of lines of seven fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify:

- minute (0-59)
- hour (0-23)
- day of the month (1-31)
- month of the year (1-12)
- day of the week (1-7 with 1 = Monday)

Each of these patterns may contain:

- a number in the range above
- two numbers separated by a minus meaning a range inclusive
- a list of numbers separated by commas meaning any of the numbers
- an asterisk meaning all legal values

The sixth field is a user name: the command will be run with that user's uid and permissions. The seventh field consists of all the text on a line following the sixth field, including spaces and tabs; this text is treated as a command which is executed by the Shell at the specified times. A percent character ("%") in this field is translated to a new-line character.

Both crontab files are checked by *cron* every minute, on the minute.

**FILES**

/usr/lib/crontab  
/usr/lib/crontab.local

**NAME**

**dcheck** – file system directory consistency check

**SYNOPSIS**

**/etc/dcheck** [ **-i** numbers ] [ filesystem ]

**DESCRIPTION**

**N.B.:** *Dcheck* is obsoleted for normal consistency checking by *fsck*(8).

*Dcheck* reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **-i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

**FILES**

Default file systems vary with installation.

**SEE ALSO**

*fsck*(8), *icheck*(8), *fs*(5), *clri*(8), *ncheck*(8)

**DIAGNOSTICS**

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

**BUGS**

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

*Dcheck* is obsoleted by *fsck* and remains for historical reasons.

**NAME**

diskpart – calculate default disk partition sizes

**SYNOPSIS**

/etc/diskpart [ -p ] [ -d ] disk-type

**DESCRIPTION**

*Diskpart* is used to calculate the disk partition sizes based on the default rules used at Berkeley. If the -p option is supplied, tables suitable for inclusion in a device driver are produced. If the -d option is supplied, an entry suitable for inclusion in the disk description file */etc/disktab* is generated; c.f. *disktab(5)*. On disks that use *bad144*-style bad-sector forwarding, space is left in the last partition on the disk for a bad sector forwarding table. The space reserved is one track for the replicated copies of the table and sufficient tracks to hold a pool of 126 sectors to which bad sectors are mapped. For more information, see *bad144(8)*.

The disk partition sizes are based on the total amount of space on the disk as given in the table below (all values are supplied in units of 512 byte sectors). The 'c' partition is, by convention, used to access the entire physical disk. The device driver tables include the space reserved for the bad sector forwarding table in the 'c' partition; those used in the *disktab* and default formats exclude reserved tracks. In normal operation, either the 'g' partition is used, or the 'd', 'e', and 'f' partitions are used. The 'g' and 'f' partitions are variable-sized, occupying whatever space remains after allocation of the fixed sized partitions. If the disk is smaller than 20 Megabytes, then *diskpart* aborts with the message "disk too small, calculate by hand".

Partition	20-60 MB	61-205 MB	206-355 MB	356+ MB
a	15884	15884	15884	15884
b	10032	33440	33440	66880
d	15884	15884	15884	15884
e	unused	55936	55936	307200
h	unused	unused	291346	291346

If an unknown disk type is specified, *diskpart* will prompt for the required disk geometry information.

**SEE ALSO**

*disktab(5)*, *bad144(8)*

**BUGS**

Certain default partition sizes are based on historical artifacts (e.g. RP06), and may result in unsatisfactory layouts.

When using the -d flag, alternate disk names are not included in the output.



**NAME**

*dmesg* – collect system diagnostic messages to form error log

**SYNOPSIS**

*/etc/dmesg* [ - ]

**DESCRIPTION**

**N.B.:** *Dmesg* is obsoleted by *syslogd*(8) for maintenance of the system error log.

*Dmesg* looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed or logged by the system when errors occur. If the - flag is given, then *dmesg* computes (incrementally) the new messages since the last time it was run and places these on the standard output.

**FILES**

*/usr/adm/msgbuf*                      scratch file for memory of - option

**SEE ALSO**

*syslogd*(8)

**NAME**

`drtest` – standalone disk test program

**DESCRIPTION**

*Drtest* is a standalone program used to read a disk track by track. It was primarily intended as a test program for new standalone drivers, but has shown useful in other contexts as well, such as verifying disks and running speed tests. For example, when a disk has been formatted (by `format(8)`), you can check that hard errors has been taken care of by running *drtest*. No hard errors should be found, but in many cases quite a few soft ECC errors will be reported.

While *drtest* is running, the cylinder number is printed on the console for every 10th cylinder read.

**EXAMPLE**

A sample run of *drtest* is shown below. In this example (using a 750), *drtest* is loaded from the root file system; usually it will be loaded from the machine's console storage device. Boldface means user input. As usual, “#” and “@” may be used to edit input.

```
>>>B/3
%%
loading hk(0,0)boot
Boot
: hk(0,0)drtest
Test program for stand-alone up and hp driver

Debugging level (1=bse, 2=ecc, 3=bse+ecc)?
Enter disk name [type(adapter,unit), e.g. hp(1,3)]? hp(0,0)
Device data: #cylinders=1024, #tracks=16, #sectors=32
Testing hp(0,0), chunk size is 16384 bytes.
(chunk size is the number of bytes read per disk access)
Start ...Make sure hp(0,0) is online
...
(errors are reported as they occur)
...
(...program restarts to allow checking other disks)
(...to abort halt machine with ^P)
```

**DIAGNOSTICS**

The diagnostics are intended to be self explanatory. Note, however, that the device number in the diagnostic messages is identified as *typeX* instead of *type(a,u)* where *X* = *a*\*8+*u*, e.g., `hp(1,3)` becomes `hp11`.

**SEE ALSO**

`format(8V)`, `bad144(8)`

**AUTHOR**

Helge Skrivervik

**NAME**

dump – incremental file system dump

**SYNOPSIS**

*/etc/dump* [ *key* [ *argument ...* ] *filesystem* ]

**DESCRIPTION**

*Dump* copies to magnetic tape all files changed after a certain date in the *filesystem*. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set 0123456789fusdWn.

- 0-9 This number is the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option 0 causes the entire filesystem to be dumped.
- f Place the dump on the next *argument* file instead of the tape. If the name of the file is "-", *dump* writes to standard output.
- u If the dump completes successfully, write the date of the beginning of the dump on file */etc/dumpdates*. This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary.
- s The size of the dump tape is specified in feet. The number of feet is taken from the next *argument*. When the specified size is reached, *dump* will wait for reels to be changed. The default tape size is 2300 feet.
- d The density of the tape, expressed in BPI, is taken from the next *argument*. This is used in calculating the amount of tape used per reel. The default is 1600.
- W *Dump* tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The W option causes *dump* to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. If the W option is set, all other options are ignored, and *dump* exits immediately.
- w Is like W, but prints only those filesystems which need to be dumped.
- n Whenever *dump* requires operator attention, notify by means similar to a *wall(1)* all of the operators in the group "operator".

If no arguments are given, the *key* is assumed to be 9u and a default file system is dumped to the default tape.

*Dump* requires operator intervention on these conditions: end of tape, end of dump, tape write error, tape open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the n key, *dump* interacts with the operator on *dump's* control terminal at times when *dump* can no longer proceed, or if something is grossly wrong. All questions *dump* poses must be answered by typing "yes" or "no", appropriately.

Since making a dump involves a lot of time and effort for full dumps, *dump* checkpoints itself at the start of each tape volume. If writing that volume fails for some reason, *dump* will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

*Dump* tells the operator what is going on at periodic intervals, including usually low estimates of the number of blocks to write, the number of tapes it will take, the time to completion, and the time to the tape change. The output is verbose, so that others know that the terminal controlling *dump* is busy, and will be for some time.

Now a short suggestion on how to perform dumps. Start with a full level 0 dump

`dump 0un`

Next, dumps of active file systems are taken on a daily basis, using a modified Tower of Hanoi algorithm, with this sequence of dump levels:

3 2 5 4 7 6 9 8 9 9 ...

For the daily dumps, a set of 10 tapes per dumped file system is used on a cyclical basis. Each week, a level 1 dump is taken, and the daily Hanoi sequence repeats with 3. For weekly dumps, a set of 5 tapes per dumped file system is used, also on a cyclical basis. Each month, a level 0 dump is taken on a set of fresh tapes that is saved forever.

#### FILES

<code>/dev/rp1g</code>	default filesystem to dump from
<code>/dev/rmt8</code>	default tape unit to dump to
<code>/etc/dumpdates</code>	new format dump date record
<code>/etc/fstab</code>	dump table: file systems and frequency
<code>/etc/group</code>	to find group <i>operator</i>

#### SEE ALSO

`restore(8)`, `dump(5)`, `fstab(5)`

#### DIAGNOSTICS

Many, and verbose.

Dump exits with zero status on success. Startup errors are indicated with an exit code of 1; abnormal termination is indicated with an exit code of 3.

#### BUGS

Fewer than 32 read errors on the filesystem are ignored. Each reel requires a new process, so parent processes for reels already written just hang around until the entire tape is written.

*Dump* with the *W* or *w* options does not report filesystems that have never been recorded in `/etc/dumpdates`, even if listed in `/etc/fstab`.

It would be nice if *dump* knew about the dump sequence, kept track of the tapes scribbled on, told the operator which tape to mount when, and provided more assistance for the operator running *restore*.

**NAME**

dumpfs – dump file system information

**SYNOPSIS**

**dumpfs** *filesystem* [*device*]

**DESCRIPTION**

*Dumpfs* prints out the super block and cylinder group information for the file system or special device specified. The listing is very long and detailed. This command is useful mostly for finding out certain file system information such as the file system block size and minimum free space percentage.

**SEE ALSO**

fs(5), disktab(5), tune2fs(8), newfs(8), fsck(8)

**NAME**

`edquota` - edit user quotas

**SYNOPSIS**

`edquota [ -p proto-user ] users...`

**DESCRIPTION**

*Edquota* is a quota editor. One or more users may be specified on the command line. For each user a temporary file is created with an ASCII representation of the current disc quotas for that user and an editor is then invoked on the file. The quotas may then be modified, new quotas added, etc. Upon leaving the editor, *edquota* reads the temporary file and modifies the binary quota files to reflect the changes made.

If the `-p` option is specified, *edquota* will duplicate the quotas of the prototypical user specified for each user specified. This is the normal mechanism used to initialize quotas for groups of users.

The editor invoked is `vi(1)` unless the environment variable `EDITOR` specifies otherwise.

Only the super-user may edit quotas.

**FILES**

<i>quotas</i>	at the root of each file system with quotas
<i>/etc/fstab</i>	to find file system names and locations

**SEE ALSO**

`quota(1)`, `quota(2)`, `quotacheck(8)`, `quotaon(8)`, `repquota(8)`

**DIAGNOSTICS**

Various messages about inaccessible files; self-explanatory.

**BUGS**

The format of the temporary file is inscrutable.

**NAME**

*fastboot*, *fasthalt* – reboot/halt the system without checking the disks

**SYNOPSIS**

*/etc/fastboot* [ *boot-options* ]

*/etc/fasthalt* [ *halt-options* ]

**DESCRIPTION**

*Fastboot* and *fasthalt* are shell scripts which reboot and halt the system without checking the file systems. This is done by creating a file */fastboot*, then invoking the *reboot* program. The system startup script, */etc/rc*, looks for this file and, if present, skips the normal invocation of *fsck*(8).

**SEE ALSO**

*halt*(8), *reboot*(8), *rc*(8)

**NAME**

*fingerd* – remote user information server

**SYNOPSIS**

*/etc/fingerd*

**DESCRIPTION**

*Fingerd* is a simple protocol based on RFC742 that provides an interface to the Name and Finger programs at several network sites. The program is supposed to return a friendly, human-oriented status report on either the system at the moment or a particular person in depth. There is no required format and the protocol consists mostly of specifying a single “command line”.

*Fingerd* listens for TCP requests at port 79. Once connected it reads a single command line terminated by a <CRLF> which is passed to *finger(1)*. *Fingerd* closes its connections as soon as the output is finished.

If the line is null (i.e. just a <CRLF> is sent) then *finger* returns a “default” report that lists all people logged into the system at that moment.

If a user name is specified (e.g. *eric*<CRLF>) then the response lists more extended information for only that particular user, whether logged in or not. Allowable “names” in the command line include both “login names” and “user names”. If a name is ambiguous, all possible derivations are returned.

**SEE ALSO**

*finger(1)*

**BUGS**

Connecting directly to the server from a TIP or an equally narrow-minded TELNET-protocol user program can result in meaningless attempts at option negotiation being sent to the server, which will foul up the command line interpretation. *Fingerd* should be taught to filter out IAC's and perhaps even respond negatively (IAC WON'T) to all option commands received.



**NAME**

format – how to format disk packs

**DESCRIPTION**

There are two ways to format disk packs. The simplest is to use the *format* program. The alternative is to use the DEC standard formatting software which operates under the DEC diagnostic supervisor. This manual page describes the operation of *format*, then concludes with some remarks about using the DEC formatter.

*Format* is a standalone program used to format and check disks prior to constructing file systems. In addition to the formatting operation, *format* records any bad sectors encountered according to DEC standard 144. Formatting is performed one track at a time by writing the appropriate headers and a test pattern and then checking the sector by reading and verifying the pattern, using the controller's ECC for error detection. A sector is marked bad if an unrecoverable media error is detected, or if a correctable ECC error too many bits in length is detected (such errors are indicated as "ECC" in the summary printed upon completing the format operation). After the entire disk has been formatted and checked, the total number of errors are reported, any bad sectors and skip sectors are marked, and a bad sector forwarding table is written to the disk in the first five even numbered sectors of the last track. It is also possible to reformat sections of the disk in units of tracks. *Format* may be used on any UNIBUS or MASSBUS drive supported by the *up* and *hp* device drivers which uses 4-byte headers (everything except RP's).

The test pattern used during the media check may be selected from one of: 0xf00f (RH750 worst case), 0xec6d (media worst case), and 0xa5a5 (alternating 1's and 0's). Normally the media worst case pattern is used.

*Format* also has an option to perform an extended "severe burn-in," which makes a number of passes using different patterns. The number of passes can be selected at run time, up to a maximum of 48, with provision for additional passes or termination after the preselected number of passes. This test runs for many hours, depending on the disk and processor.

Each time *format* is run to format an entire disk, a completely new bad sector table is generated based on errors encountered while formatting. The device driver, however, will always attempt to read any existing bad sector table when the device is first opened. Thus, if a disk pack has never previously been formatted, or has been formatted with different sectoring, five error messages will be printed when the driver attempts to read the bad sector table; these diagnostics should be ignored.

Formatting a 400 megabyte disk on a MASSBUS disk controller usually takes about 20 minutes. Formatting on a UNIBUS disk controller takes significantly longer. For every hundredth cylinder formatted *format* prints a message indicating the current cylinder being formatted. (This message is just to reassure people that nothing is amiss.)

*Format* uses the standard notation of the standalone I/O library in identifying a drive to be formatted. A drive is specified as *zz(x,y)*, where *zz* refers to the controller type (either *hp* or *up*), *x* is the unit number of the drive; 8 times the UNIBUS or MASSBUS adaptor number plus the MASSBUS drive number or UNIBUS drive unit number; and *y* is the file system partition on drive *x* (this should always be 0). For example, "*hp(1,0)*" indicates that drive 1 on MASSBUS adaptor 0 should be formatted; while "*up(10,0)*" indicates that UNIBUS drive 2 on UNIBUS adaptor 1 should be formatted.

Before each formatting attempt, *format* prompts the user in case debugging should be enabled in the appropriate device driver. A carriage return disables debugging information.

*Format* should be used prior to building file systems (with *newfs(8)*) to insure that all sectors with uncorrectable media errors are remapped. If a drive develops uncorrectable defects after formatting, either *bad144(8)* or *badsect(8)* should be able to avoid the bad sectors.

**EXAMPLE**

A sample run of *format* is shown below. In this example (using a VAX-11/780), *format* is loaded from the console floppy; on an 11/750 *format* will be loaded from the root file system with *boot*(8) following a "B/3" command. Boldface means user input. As usual, "@" and "@" may be used to edit input.

```
>>>L FORMAT
LOAD DONE, 00004400 BYTES LOADED
>>>S 2
Disk format/check utility

Enable debugging (0=none, 1=bse, 2=ecc, 3=bse+ecc)? 0
Device to format? hp(8,0)
(error messages may occur as old bad sector table is read)
Formatting drive hp0 on adaptor 1: verify (yes/no)? yes
Device data: #cylinders=842, #tracks=20, #sectors=48
Starting cylinder (0):
Starting track (0):
Ending cylinder (841):
Ending track (19):
Available test patterns are:
    1 - (f00f) RH750 worst case
    2 - (ec6d) media worst case
    3 - (a5a5) alternating 1's and 0's
    4 - (ffff) Severe burnin (up to 48 passes)
Pattern (one of the above, other to restart)? 2
Maximum number of bit errors to allow for soft ECC (3):
Start formatting...make sure the drive is online
...
(soft ecc's and other errors are reported as they occur)
...
(if 4 write check errors were found, the program terminates like this...)
...
Errors:
Bad sector: 0
Write check: 4
Hard ECC: 0
Other hard: 0
Marked bad: 0
Skipped: 0
Total of 4 hard errors revectorred.
Writing bad sector table at block 808272
(808272 is the block # of the first block in the bad sector table)
Done
(...program restarts to allow formatting other disks)
(...to abort halt machine with ^P)
```

**DIAGNOSTICS**

The diagnostics are intended to be self explanatory.

**USING DEC SOFTWARE TO FORMAT**

**Warning:** These instructions are for people with 11/780 CPU's. The steps needed for 11/750 or 11/730 cpu's are similar, but not covered in detail here.

The formatting procedures are different for each type of disk. Listed here are the formatting procedures for RK07's, RP0X, and RM0X disks.

You should shut down UNIX and halt the machine to do any disk formatting. Make certain you put in the pack you want formatted. It is also a good idea to spin down or write protect the disks you don't want to format, just in case.

**Formatting an RK07.** Load the console floppy labeled, "RX11 VAX DSK LD DEV #1" in the console disk drive, and type the following commands:

```
>>>BOOT
DIAGNOSTIC SUPERVISOR. ZZ-ESSAA-X5.0-119 23-JAN-1980 12:44:40.03
DS>ATTACH DW780 SBI DW0 3 5
DS>ATTACH RK611 DMA
DS>ATTACH RK07 DW0 DMA0
DS>SELECT DMA0
DS>LOAD EVRAC
DS>START/SEC:PACKINIT
```

**Formatting an RP0X.** Follow the above procedures except that the ATTACH and SELECT lines should read:

```
DS>ATTACH RH780 SBI RH0 8 5
DS>ATTACH RP0X RH0 DBA0(RP0X is, e.g. RP06)
DS>SELECT DBA0
```

This is for drive 0 on mba0; use 9 instead of 8 for mba1, etc.

**Formatting an RM0X.** Follow the above procedures except that the ATTACH and SELECT lines should read:

```
DS>ATTACH RH780 SBI RH0 8 5
DS>ATTACH RM0X RH0 DRA0
DS>SELECT DRA0
```

Don't forget to put your UNIX console floppy back in the floppy disk drive.

#### SEE ALSO

bad144(8), badsect(8), newfs(8)

#### BUGS

An equivalent facility should be available which operates under a running UNIX system.

It should be possible to reformat or verify part or all of a disk, then update the existing bad sector table.

**NAME**

`fsck` - file system consistency check and interactive repair

**SYNOPSIS**

```
/etc/fsck -p [ filesystem ... ]
/etc/fsck [ -b block# ] [ -y ] [ -n ] [ filesystem ] ...
```

**DESCRIPTION**

The first form of *fsck* preens a standard set of filesystems or the specified file systems. It is normally used in the script `/etc/rc` during automatic reboot. In this case *fsck* reads the table `/etc/fstab` to determine which file systems to check. It uses the information there to inspect groups of disks in parallel taking maximum advantage of i/o overlap to check the file systems as quickly as possible. Normally, the root file system will be checked on pass 1, other "root" ("a" partition) file systems on pass 2, other small file systems on separate passes (e.g. the "d" file systems on pass 3 and the "e" file systems on pass 4), and finally the large user file systems on the last pass, e.g. pass 5. Only partitions in `fstab` that are mounted "rw" or "rq" and that have non-zero pass number are checked.

The system takes care that only a restricted class of innocuous inconsistencies can happen unless hardware or software failures intervene. These are limited to the following:

- Unreferenced inodes
- Link counts in inodes too large
- Missing blocks in the free list
- Blocks in the free list also in files
- Counts in the super-block wrong

These are the only inconsistencies that *fsck* with the `-p` option will correct; if it encounters other inconsistencies, it exits with an abnormal return status and an automatic reboot will then fail. For each corrected inconsistency one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. After successfully correcting a file system, *fsck* will print the number of files on that file system, the number of used and free blocks, and the percentage of fragmentation.

If sent a QUIT signal, *fsck* will finish the file system checks, then exit with an abnormal return status that causes the automatic reboot to fail. This is useful when you wish to finish the file system checks, but do not want the machine to come up multiuser.

Without the `-p` option, *fsck* audits and interactively repairs inconsistent conditions for file systems. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that some of the corrective actions which are not correctable under the `-p` option will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond yes or no. If the operator does not have write permission on the file system *fsck* will default to a `-n` action.

*Fsck* has more consistency checks than its predecessors *check*, *dcheck*, *fcheck*, and *icheck* combined.

The following flags are interpreted by *fsck*.

- `-b` Use the block specified immediately after the flag as the super block for the file system. Block 32 is always an alternate super block.
- `-y` Assume a yes response to all questions asked by *fsck*; this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

- n Assume a no response to all questions asked by *fsck*; do not open the file system for writing.

If no filesystems are given to *fsck* then a default list of file systems is read from the file */etc/fstab*.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
  - Directory size not of proper format.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
  - File pointing to unallocated inode.
  - Inode number out of range.
8. Super Block checks:
  - More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the *lost+found* directory. The name assigned is the inode number. If the *lost+found* directory does not exist, it is created. If there is insufficient space its size is increased.

Checking the raw device is almost always faster.

#### FILES

*/etc/fstab* contains default list of file systems to check.

#### DIAGNOSTICS

The diagnostics produced by *fsck* are fully enumerated and explained in Appendix A of "Fscck - The UNIX File System Check Program" (SMM:5).

#### SEE ALSO

*fstab(5)*, *fs(5)*, *newfs(8)*, *mkfs(8)*, *crash(8V)*, *reboot(8)*

#### BUGS

There should be some way to start a *fsck -p* at pass *n*.

# NAME

ftpd - DARPA Internet File Transfer Protocol server

# SYNOPSIS

/etc/ftpd [ -d ] [ -l ] [ -ttimeout ]

# DESCRIPTION

*Ftpd* is the DARPA Internet File Transfer Protocol server process. The server uses the TCP protocol and listens at the port specified in the "ftp" service specification; see *services(5)*.

If the -d option is specified, debugging information is written to the syslog.

If the -l option is specified, each ftp session is logged in the syslog.

The ftp server will timeout an inactive session after 15 minutes. If the -t option is specified, the inactivity timeout period will be set to *timeout*.

The ftp server currently supports the following ftp requests; case is not distinguished.

Request	Description
ABOR	abort previous command
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CDUP	change to parent of current working directory
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory ("ls -lg")
MKD	make a directory
MODE	specify data transfer <i>mode</i>
NLST	give name list of files in directory ("ls")
NOOP	do nothing
PASS	specify password
PASV	prepare for server-to-server transfer
PORT	specify data connection port
PWD	print the current working directory
QUIT	terminate session
RETR	retrieve a file
RMD	remove a directory
RNFR	specify rename-from file name
RNTO	specify rename-to file name
STOR	store a file
STOU	store a file with a unique name
STRU	specify data transfer <i>structure</i>
TYPE	specify data transfer <i>type</i>
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining ftp requests specified in Internet RFC 959 are recognized, but not implemented.

The ftp server will abort an active file transfer only when the ABOR command is preceded by a Telnet "Interrupt Process" (IP) signal and a Telnet "Synch" signal in the command Telnet stream, as described in Internet RFC 959.

*Ftpd* interprets file names according to the "globbing" conventions used by *cs(1)*. This allows users to utilize the metacharacters `"*?[]()"`.

*Ftpd* authenticates users according to three rules.

- 1) The user name must be in the password data base, */etc/passwd*, and not have a null password. In this case a password must be provided by the client before any file operations may be performed.
- 2) The user name must not appear in the file */etc/ftpusers*.
- 3) The user must have a standard shell returned by *getusershell(3)*.
- 4) If the user name is "anonymous" or "ftp", an anonymous ftp account must be present in the password file (user "ftp"). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

In the last case, *ftpd* takes special measures to restrict the client's access privileges. The server performs a *chroot(2)* command to the home directory of the "ftp" user. In order that system security is not breached, it is recommended that the "ftp" subtree be constructed with care; the following rules are recommended.

`~ftp`) Make the home directory owned by "ftp" and unwritable by anyone.

`~ftp/bin`)

Make this directory owned by the super-user and unwritable by anyone. The program *ls(1)* must be present to support the list commands. This program should have mode 111.

`~ftp/etc`)

Make this directory owned by the super-user and unwritable by anyone. The files *passwd(5)* and *group(5)* must be present for the *ls* command to work properly. These files should be mode 444.

`~ftp/pub`)

Make this directory mode 777 and owned by "ftp". Users should then place files which are to be accessible via the anonymous account in this directory.

#### SEE ALSO

*ftp(1C)*, *getusershell(3)*, *syslogd(8)*

#### BUGS

The anonymous account is inherently dangerous and should avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user id of the logged in user, reverting to the super-user only when binding addresses to sockets. The possible security holes have been extensively scrutinized, but are possibly incomplete.

**NAME**

`gettable` – get NIC format host tables from a host

**SYNOPSIS**

`/etc/gettable [ -v ] host [ outfile ]`

**DESCRIPTION**

*Gettable* is a simple program used to obtain the NIC standard host tables from a “nickname” server. The indicated *host* is queried for the tables. The tables, if retrieved, are placed in the file *outfile* or by default, *hosts.txt*.

The `-v` option will get just the version number instead of the complete host table and put the output in the file *outfile* or by default, *hosts.ver*.

*Gettable* operates by opening a TCP connection to the port indicated in the service specification for “nickname”. A request is then made for “ALL” names and the resultant information is placed in the output file.

*Gettable* is best used in conjunction with the *htable(8)* program which converts the NIC standard file format to that used by the network library lookup routines.

**SEE ALSO**

`intro(3N)`, `htable(8)`, `named(8)`

**BUGS**

If the name-domain system provided network name mapping well as host name mapping, *gettable* would no longer be needed.



**NAME**

`getty` - set terminal mode

**SYNOPSIS**

`/etc/getty [ type [ tty ] ]`

**DESCRIPTION**

*Getty* is usually invoked by *init*(8) to open and initialize the tty line, read a login name, and invoke *login*(1). *getty* attempts to adapt the system to the speed and type of terminal being used.

The argument *tty* is the special device file in */dev* to open for the terminal (e.g., "ttyh0"). If there is no argument or the argument is "-", the tty line is assumed to be open as file descriptor 0.

The *type* argument can be used to make *getty* treat the terminal line specially. This argument is used as an index into the *gettytab*(5) database, to determine the characteristics of the line. If there is no argument, or there is no such table, the default table is used. If there is no */etc/gettytab* a set of system defaults is used. If indicated by the table located, *getty* will clear the terminal screen, print a banner heading, and prompt for a login name. Usually either the banner of the login prompt will include the system hostname. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the 'break' ('interrupt') key. The speed is usually then changed and the 'login:' is typed again; a second 'break' changes the speed again and the 'login:' is typed once more. Successive 'break' characters cycle through the same standard set of speeds.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *tty*(4)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, *login* is called with the user's name as an argument.

Most of the default actions of *getty* can be circumvented, or modified, by a suitable *gettytab* table.

*Getty* can be set to timeout after some interval, which will cause dial up lines to hang up if the login name is not entered reasonably quickly.

**DIAGNOSTICS**

*ttysx*: No such device or address. *ttysx*: No such file or address. A terminal which is turned on in the *ttys* file cannot be opened, likely because the requisite lines are either not configured into the system, the associated device was not attached during boot-time system configuration, or the special file in */dev* does not exist.

**FILES**

*/etc/gettytab*

**SEE ALSO**

*gettytab*(5), *init*(8), *login*(1), *ioctl*(2), *tty*(4), *ttys*(5)

**NAME**

halt - stop the processor

**SYNOPSIS**

/etc/halt [ -n ] [ -q ] [ -y ]

**DESCRIPTION**

*Halt* writes out sandbagged information to the disks and then stops the processor. The machine does not reboot, even if the auto-reboot switch is set on the console.

The -n option prevents the sync before stopping. The -q option causes a quick halt, no graceful shutdown is attempted. The -y option is needed if you are trying to halt the system from a dialup.

*Halt* normally logs the shutdown using *syslog*(8) and places a shutdown record in the login accounting file */usr/adm/wtmp*. These actions are inhibited if the -n or -q options are present.

**SEE ALSO**

reboot(8), shutdown(8), syslogd(8)

**BUGS**

It is very difficult to halt a VAX, as the machine wants to then reboot itself. A rather tight loop suffices.

**NAME**

*htable* - convert NIC standard format host tables

**SYNOPSIS**

*/etc/htable* [ *-c connected-nets* ] [ *-l local-nets* ] *file*

**DESCRIPTION**

*Htable* is used to convert host files in the format specified in Internet RFC 810 to the format used by the network library routines. Three files are created as a result of running *htable*: *hosts*, *networks*, and *gateways*. The *hosts* file may be used by the *gethostbyname*(3N) routines in mapping host names to addresses if the nameserver, *named*(8), is not used. The *networks* file is used by the *getnetent*(3N) routines in mapping network names to numbers. The *gateways* file may be used by the routing daemon in identifying "passive" Internet gateways; see *routed*(8C) for an explanation.

If any of the files *localhosts*, *localnetworks*, or *localgateways* are present in the current directory, the file's contents is prepended to the output file. Of these, only the *gateways* file is interpreted. This allows sites to maintain local aliases and entries which are not normally present in the master database. Only one gateway to each network will be placed in the *gateways* file; a gateway listed in the *localgateways* file will override any in the input file.

If the *gateways* file is to be used, a list of networks to which the host is directly connected is specified with the *-c* flag. The networks, separated by commas, may be given by name or in Internet-standard dot notation, e.g. *-c arpanet,128.32,local-ether-net*. *Htable* only includes gateways which are directly connected to one of the networks specified, or which can be reached from another gateway on a connected net.

If the *-l* option is given with a list of networks (in the same format as for *-c*), these networks will be treated as "local," and information about hosts on local networks is taken only from the *localhosts* file. Entries for local hosts from the main database will be omitted. This allows the *localhosts* file to completely override any entries in the input file.

*Htable* is best used in conjunction with the *gettable*(8C) program which retrieves the NIC database from a host.

**SEE ALSO**

*intro*(3N), *gettable*(8C), *named*(8)

**BUGS**

If the name-domain system provided network name mapping well as host name mapping, *htable* would no longer be needed.

**NAME**

*icheck* - file system storage consistency check

**SYNOPSIS**

*/etc/icheck* [ -s ] [ -b numbers ] [ filesystem ]

**DESCRIPTION**

**N.B.:** *Icheck* is obsolete for normal consistency checking by *fsck*(8).

*Icheck* examines a file system, builds a hit map of used blocks, and compares this hit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of *icheck* includes a report of

The total number of files and the numbers of regular, directory, block special and character special files.

The total number of blocks in use and the numbers of single-, double-, and triple-indirect blocks and directory blocks.

The number of free blocks.

The number of blocks missing; i.e. not in any file nor in the free list.

The -s option causes *icheck* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, had in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been corrupted these words will have to be patched. The -s option causes the normal output reports to be suppressed.

Following the -b option is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

*Icheck* is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

**FILES**

Default file systems vary with installation.

**SEE ALSO**

*fsck*(8), *dcheck*(8), *ncheck*(8), *fs*(5), *clri*(8)

**DIAGNOSTICS**

For duplicate blocks and bad blocks (which lie outside the file system) *icheck* announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and *icheck* considers it to contain 0. 'Bad freeblock' means that a block number outside the available space was encountered in the free list. 'n dups in free' means that n blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

**BUGS**

Since *icheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

The system should be fixed so that the reboot after fixing the root file system is not necessary.

**NAME**

`ifconfig` – configure network interface parameters

**SYNOPSIS**

```
/etc/ifconfig interface address_family [ address [ dest_address ] ] [ parameters ]
/etc/ifconfig interface [ protocol_family ]
```

**DESCRIPTION**

*Ifconfig* is used to assign an address to a network interface and/or configure network interface parameters. *Ifconfig* must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address or other operating parameters. The *interface* parameter is a string of the form "name unit", e.g. "en0".

Since an interface may receive transmissions in differing protocols, each of which may require separate naming schemes, it is necessary to specify the *address\_family*, which may change the interpretation of the remaining parameters. The address families currently supported are "inet" and "ns".

For the DARPA-Internet family, the address is either a host name present in the host name data base, *hosts(5)*, or a DARPA Internet address expressed in the Internet standard "dot notation". For the Xerox Network Systems(tm) family, addresses are *net:a.b.c.d.e.f*, where *net* is the assigned network number (in decimal), and each of the six bytes of the host number, *a* through *f*, are specified in hexadecimal. The host number may be omitted on 10Mb/s Ethernet interfaces, which use the hardware physical address, and on interfaces other than the first.

The following parameters may be set with *ifconfig*:

- |                        |  |
|------------------------|--|
| <b>up</b>              | Mark an interface "up". This may be used to enable an interface after an "ifconfig down." It happens automatically when setting the first address on an interface. If the interface was reset when previously marked down, the hardware will be re-initialized.  |
| <b>down</b>            | Mark an interface "down". When an interface is marked "down", the system will not attempt to transmit messages through that interface. If possible, the interface will be reset to disable reception as well. This action does not automatically disable routes using the interface.   |
| <b>trailers</b>        | Request the use of a "trailer" link level encapsulation when sending (default). If a network interface supports <i>trailers</i> , the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver. On networks that support the Address Resolution Protocol (see <i>arp(4P)</i> ; currently, only 10 Mb/s Ethernet), this flag indicates that the system should request that other systems use trailers when sending to this host. Similarly, trailer encapsulations will be sent to other hosts that have made such requests. Currently used by Internet protocols only. |
| <b>-trailers</b>       | Disable the use of a "trailer" link level encapsulation.   |
| <b>arp</b>             | Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addresses and 10Mb/s Ethernet addresses.   |
| <b>-arp</b>            | Disable the use of the Address Resolution Protocol.  |
| <b>metric <i>n</i></b> | Set the routing metric of the interface to <i>n</i> , default 0. The routing metric is used by the routing protocol ( <i>routed(8c)</i> ). Higher metrics have the effect of making a route less favorable; metrics are counted as addition hops to the destination network or host.   |

<b>debug</b>	Enable driver dependent debugging code; usually, this turns on extra console error logging.
<b>-debug</b>	Disable driver dependent debugging code.
<b>netmask <i>mask</i></b>	(Inet only) Specify how much of the address to reserve for subdividing networks into sub-networks. The mask includes the network part of the local address and the subnet part, which is taken from the host field of the address. The mask can be specified as a single hexadecimal number with a leading 0x, with a dot-notation Internet address, or with a pseudo-network name listed in the network table <i>networks(5)</i> . The mask contains 1's for the bit positions in the 32-bit address which are to be used for the network and subnet parts, and 0's for the host part. The mask should contain at least the standard network portion, and the subnet field should be contiguous with the network portion.
<b>dstaddr</b>	Specify the address of the correspondent on the other end of a point to point link.
<b>broadcast</b>	(Inet only) Specify the address to use to represent broadcasts to the network. The default broadcast address is the address with a host part of all 1's.
<b>ipdst</b>	(NS only) This is used to specify an Internet host who is willing to receive ip packets encapsulating NS packets bound for a remote network. In this case, an apparent point to point link is constructed, and the address specified will be taken as the NS address and network of the destinee.

*Ifconfig* displays the current configuration for a network interface when no optional parameters are supplied. If a protocol family is specified, *Ifconfig* will report only the details specific to that protocol family.

Only the super-user may modify the configuration of a network interface.

#### DIAGNOSTICS

Messages indicating the specified interface does not exist, the requested address is unknown, or the user is not privileged and tried to alter an interface's configuration.

#### SEE ALSO

*netstat(1)*, *intro(4N)*, *rc(8)*

**NAME**

**implog** – IMP log interpreter

**SYNOPSIS**

**/etc/implog** [ **-D** ] [ **-f** ] [ **-c** ] [ **-r** ] [ **-l** *link* ] [ **-h** *host#* ] [ **-i** *imp#* ] [ **-t** *message-type* ]

**DESCRIPTION**

*Implog* is program which interprets the message log produced by *implogd*(8C).

If no arguments are specified, *implog* interprets and prints every message present in the message file. Options may be specified to force printing only a subset of the logged messages.

- D** Do not show data messages.
- f** Follow the logging process in action. This flag causes *implog* to print the current contents of the log file, then check for new logged messages every 5 seconds.
- c** In addition to printing any data messages logged, show the contents of the data in hexadecimal bytes.
- r** Print the raw imp leader, showing all fields, in addition to the formatted interpretation according to type.
- l** *link#* Show only those messages received on the specified "link". If no value is given for the link, the link number of the IP protocol is assumed.
- h** *host#* Show only those messages received from the specified host. (Usually specified in conjunction with an *imp*.)
- i** *imp#* Show only those messages received from the specified *imp*.
- t** *message-type* Show only those messages received of the specified message type.

**SEE ALSO**

*imp*(4P), *implogd*(8C)

**BUGS**

Can not specify multiple hosts, *imps*, etc. Can not follow reception of messages without looking at those currently in the file.

**NAME**

implogd – IMP logger process

**SYNOPSIS**

/etc/implogd [ -d ]

**DESCRIPTION**

*Implogd* is program which logs error messages from the IMP, placing them in the file */usr/adm/implog*.

Entries in the file are variable length. Each log entry has a fixed length header of the form:

```
struct sockstamp (  
    short  sin_family;  
    u_short sin_port;  
    struct  in_addr sin_addr;  
    time_t  sin_time;  
    int     sin_len;  
);
```

followed, possibly, by the message received from the IMP. Each time the logging process is started up it places a time stamp entry in the file (a header with *sin\_len* field set to 0).

The logging process will catch only those message from the IMP which are not processed by a protocol module, e.g. IP. This implies the log should contain only status information such as "IMP going down" messages, "host down" and other error messages, and, perhaps, stray NCP messages.

**SEE ALSO**

imp(4P), implog(8C)



**NAME**

inetd – internet “super-server”

**SYNOPSIS**

/etc/inetd [ -d ] [ configuration file ]

**DESCRIPTION**

*Inetd* should be run at boot time by */etc/rc.local*. It then listens for connections on certain internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket (except in some cases which will be described below). Essentially, *inetd* allows running one daemon to invoke several others, reducing load on the system.

Upon execution, *inetd* reads its configuration information from a configuration file which, by default, is */etc/inetd.conf*. There must be an entry for each field of the configuration file, with entries for each field separated by a tab or a space. Comments are denoted by a “#” at the beginning of a line. There must be an entry for each field. The fields of the configuration file are as follows:

- service name
- socket type
- protocol
- wait/nowait
- user
- server program
- server program arguments

The *service name* entry is the name of a valid service in the file */etc/services*/. For “internal” services (discussed below), the service name *must* be the official name of the service (that is, the first entry in */etc/services*).

The *socket type* should be one of “stream”, “dgram”, “raw”, “rdm”, or “seqpacket”, depending on whether the socket is a stream, datagram, raw, reliably delivered message, or sequenced packet socket.

The *protocol* must be a valid protocol as given in */etc/protocols*. Examples might be “tcp” or “udp”.

The *wait/nowait* entry is applicable to datagram sockets only (other sockets should have a “nowait” entry in this space). If a datagram server connects to its peer, freeing the socket so *inetd* can receive further messages on the socket, it is said to be a “multi-threaded” server, and should use the “nowait” entry. For datagram servers which process all incoming datagrams on a socket and eventually time out, the server is said to be “single-threaded” and should use a “wait” entry. “Comsat” (“biff”) and “talk” are both examples of the latter type of datagram server. *Tftpd* is an exception; it is a datagram server that establishes pseudo-connections. It must be listed as “wait” in order to avoid a race; the server reads the first packet, creates a new socket, and then forks and exits to allow *inetd* to check for new service requests to spawn new servers.

The *user* entry should contain the user name of the user as whom the server should run. This allows for servers to be given less permission than root. The *server program* entry should contain the pathname of the program which is to be executed by *inetd* when a request is found on its socket. If *inetd* provides this service internally, this entry should be “internal”.

The arguments to the server program should be just as they normally are, starting with *argv[0]*, which is the name of the program. If the service is provided internally, the word “internal” should take the place of this entry.

*Inetd* provides several "trivial" services internally by use of routines within itself. These services are "echo", "discard", "chargen" (character generator), "daytime" (human readable time), and "time" (machine readable time, in the form of the number of seconds since midnight, January 1, 1900). All of these services are tcp based. For details of these services, consult the appropriate RFC from the Network Information Center.

*Inetd* rereads its configuration file when it receives a hangup signal, SIGHUP. Services may be added, deleted or modified when the configuration file is reread.

**SEE ALSO**

comsat(8C), ftpd(8C), rexecd(8C), rlogind(8C), rshd(8C), telnetd(8C), tftpd(8C)

**NAME**

init – process control initialization

**SYNOPSIS**

/etc/init

**DESCRIPTION**

*Init* is invoked inside UNIX as the last step in the boot procedure. It normally then runs the automatic reboot sequence as described in *reboot*(8), and if this succeeds, begins multi-user operation. If the reboot fails, it commences single user operation by giving the super-user a shell on the console. It is possible to pass parameters from the boot program to *init* so that single user operation is commenced immediately. When such single user operation is terminated by killing the single-user shell (i.e. by hitting ^D), *init* runs */etc/rc* without the reboot parameter. This command file performs housekeeping operations such as removing temporary files, mounting file systems, and starting daemons.

In multi-user operation, *init*'s role is to create a process for each terminal port on which a user may log in. To begin such operations, it reads the file */etc/tty*s and executes a command for each terminal specified in the file. This command will usually be */etc/getty*. *Getty* opens and initializes the terminal line, reads the user's name and invokes *login* to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. The *wtmp* entry is made only if a user logged in successfully on the line. Then the appropriate terminal is reopened and *getty* is reinvoked.

*Init* catches the *hangup* signal (signal SIGHUP) and interprets it to mean that the file */etc/tty*s should be read again. The Shell process on each line which used to be active in *ttys* but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add terminal lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use 'kill -HUP 1.'

*Init* will terminate multi-user operations and resume single-user mode if sent a terminate (TERM) signal, i.e. "kill -TERM 1". If there are processes outstanding which are deadlocked (due to hardware or software failure), *init* will not wait for them all to die (which might take forever), but will time out after 30 seconds and print a warning message.

*Init* will cease creating new *getty*'s and allow the system to slowly die away, if it is sent a terminal stop (TSTP) signal, i.e. "kill -TSTP 1". A later hangup will resume full multi-user operations, or a terminate will initiate a single user shell. This hook is used by *reboot*(8) and *halt*(8).

*Init*'s role is so critical that if it dies, the system will reboot itself automatically. If, at bootstrap time, the *init* process cannot be located, the system will loop in user mode at location 0x13.

**DIAGNOSTICS**

*/etc/getty* *gettyargs* failing, sleeping. A process being started to service a line is exiting quickly each time it is started. This is often caused by a ringing or noisy terminal line. *Init* will sleep for 30 seconds,

**WARNING:** Something is hung (wont die); ps axl advised. A process is hung and could not be killed when the system was shutting down. This is usually caused by a process which is stuck in a device driver due to a persistent device error condition.

**FILES**

/dev/console, /dev/tty\*, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

**SEE ALSO**

login(1), kill(1), sh(1), ttys(5), crash(8V), getty(8), rc(8), reboot(8), halt(8), shutdown(8)

**NAME**

**kgmon** – generate a dump of the operating system's profile buffers

**SYNOPSIS**

**/etc/kgmon** [ **-b** ] [ **-h** ] [ **-r** ] [ **-p** ] [ **system** ] [ **memory** ]

**DESCRIPTION**

*Kgmon* is a tool used when profiling the operating system. When no arguments are supplied, *kgmon* indicates the state of operating system profiling as running, off, or not configured. (see *config(8)*) If the **-p** flag is specified, *kgmon* extracts profile data from the operating system and produces a *gmon.out* file suitable for later analysis by *gprof(1)*.

The following options may be specified:

- b** Resume the collection of profile data.
- h** Stop the collection of profile data.
- p** Dump the contents of the profile buffers into a *gmon.out* file.
- r** Reset all the profile buffers. If the **-p** flag is also specified, the *gmon.out* file is generated before the buffers are reset.

If neither **-b** nor **-h** is specified, the state of profiling collection remains unchanged. For example, if the **-p** flag is specified and profile data is being collected, profiling will be momentarily suspended, the operating system profile buffers will be dumped, and profiling will be immediately resumed.

**FILES**

**/vmunix** – the default system  
**/dev/kmem** – the default memory

**SEE ALSO**

*gprof(1)*, *config(8)*

**DIAGNOSTICS**

Users with only read permission on **/dev/kmem** cannot change the state of profiling collection. They can get a *gmon.out* file with the warning that the data may be inconsistent if profiling is in progress.

**NAME**

*lpc* -- line printer control program

**SYNOPSIS**

*/etc/lpc* [ command [ argument ... ] ]

**DESCRIPTION**

*Lpc* is used by the system administrator to control the operation of the line printer system. For each line printer configured in */etc/printcap*, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer daemons.

Without any arguments, *lpc* will prompt for commands from the standard input. If arguments are supplied, *lpc* interprets the first argument as a command and the remaining arguments as parameters to the command. The standard input may be redirected causing *lpc* to read commands from file. Commands may be abbreviated; the following is the list of recognized commands.

? [ command ... ]

help [ command ... ]

Print a short description of each command specified in the argument list, or, if no arguments are given, a list of the recognized commands.

abort { all | printer ... }

Terminate an active spooling daemon on the local host immediately and then disable printing (preventing new daemons from being started by *lpr*) for the specified printers.

clean { all | printer ... }

Remove any temporary files, data files, and control files that cannot be printed (i.e., do not form a complete printer job) from the specified printer queue(s) on the local machine.

disable { all | printer ... }

Turn the specified printer queues off. This prevents new printer jobs from being entered into the queue by *lpr*.

down { all | printer } message ...

Turn the specified printer queue off, disable printing and put *message* in the printer status file. The message doesn't need to be quoted, the remaining arguments are treated like *echo(1)*. This is normally used to take a printer down and let others know why (*lpq* will indicate the printer is down and print the status message).

enable { all | printer ... }

Enable spooling on the local queue for the listed printers. This will allow *lpr* to put new jobs in the spool queue.

exit

quit

Exit from *lpc*.

restart { all | printer ... }

Attempt to start a new printer daemon. This is useful when some abnormal condition causes the daemon to die unexpectedly leaving jobs in the queue. *Lpq* will report that there is no daemon present when this condition occurs. If the user is the super-user, try to abort the current daemon first (i.e., kill and restart a stuck daemon).

**start** { all | printer ... }  
    Enable printing and start a spooling daemon for the listed printers.

**status** { all | printer ... }  
    Display the status of daemons and queues on the local machine.

**stop** { all | printer ... }  
    Stop a spooling daemon after the current job completes and disable printing.

**topq printer** [ jobnum ... ] [ user ... ]  
    Place the jobs in the order listed at the top of the printer queue.

**up** { all | printer ... }  
    Enable everything and start a new printer daemon. Undoes the effects of *down*.

**FILES**

/etc/printcap	printer description file
/usr/spool/*	spool directories
/usr/spool/*lock	lock file for queue control

**SEE ALSO**

lpd(8), lpr(1), lpq(1), lprm(1), printcap(5)

**DIAGNOSTICS**

?Ambiguous command	abbreviation matches more than one command
?Invalid command	no match was found
?Privileged command	command can be executed by root only

**NAME**

*lpd* – line printer daemon

**SYNOPSIS**

*/usr/lib/lpd* [ -l ] [ port # ]

**DESCRIPTION**

*Lpd* is the line printer daemon (spool area handler) and is normally invoked at boot time from the *rc(8)* file. It makes a single pass through the *printcap(5)* file to find out about the existing printers and prints any files left after a crash. It then uses the system calls *listen(2)* and *accept(2)* to receive requests to print files in the queue, transfer files to the spooling area, display the queue, or remove jobs from the queue. In each case, it forks a child to handle the request so the parent can continue to listen for more requests. The Internet port number used to rendezvous with other processes is normally obtained with *getservbyname(3)* but can be changed with the *port#* argument. The *-l* flag causes *lpd* to log valid requests received from the network. This can be useful for debugging purposes.

Access control is provided by two means. First, All requests must come from one of the machines listed in the file */etc/hosts.equiv* or */etc/hosts.lpd*. Second, if the “rs” capability is specified in the *printcap* entry for the printer being accessed, *lpr* requests will only be honored for those users with accounts on the machine with the printer.

The file *minfree* in each spool directory contains the number of disk blocks to leave free so that the line printer queue won't completely fill the disk. The *minfree* file can be edited with your favorite text editor.

The file *lock* in each spool directory is used to prevent multiple daemons from becoming active simultaneously, and to store information about the daemon process for *lpr(1)*, *lpq(1)*, and *lprm(1)*. After the daemon has successfully set the lock, it scans the directory for files beginning with *cf*. Lines in each *cf* file specify files to be printed or non-printing actions to be performed. Each such line begins with a key character to specify what to do with the remainder of the line.

- J     Job Name. String to be used for the job name on the burst page.
- C     Classification. String to be used for the classification line on the burst page.
- L     Literal. The line contains identification info from the password file and causes the banner page to be printed.
- T     Title. String to be used as the title for *pr(1)*.
- H     Host Name. Name of the machine where *lpr* was invoked.
- P     Person. Login name of the person who invoked *lpr*. This is used to verify ownership by *lprm*.
- M     Send mail to the specified user when the current print job completes.
- f     Formatted File. Name of a file to print which is already formatted.
- l     Like “f” but passes control characters and does not make page breaks.
- p     Name of a file to print using *pr(1)* as a filter.
- t     Troff File. The file contains *troff(1)* output (cat phototypesetter commands).
- n     Dittroff File. The file contains device independent troff output.
- d     DVI File. The file contains *Tex(1)* output (DVI format from Stanford).
- g     Graph File. The file contains data produced by *plot(3X)*.
- c     Cifplot File. The file contains data produced by *cifplot*.



- v     The file contains a raster image.
- r     The file contains text data with FORTRAN carriage control characters.
- 1     Troff Font R. Name of the font file to use instead of the default.
- 2     Troff Font I. Name of the font file to use instead of the default.
- 3     Troff Font B. Name of the font file to use instead of the default.
- 4     Troff Font S. Name of the font file to use instead of the default.
- W     Width. Changes the page width (in characters) used by *pr*(1) and the text filters.
- I     Indent. The number of characters to indent the output by (in ascii).
- U     Unlink. Name of file to remove upon completion of printing.
- N     File name. The name of the file which is being printed, or a blank for the standard input (when *lpr* is invoked in a pipeline).

If a file can not be opened, a message will be logged via *syslog*(3) using the *LOG\_LPR* facility. *Lpd* will try up to 20 times to reopen a file it expects to be there, after which it will skip the file to be printed.

*Lpd* uses *flock*(2) to provide exclusive access to the lock file and to prevent multiple daemons from becoming active simultaneously. If the daemon should be killed or die unexpectedly, the lock file need not be removed. The lock file is kept in a readable ASCII form and contains two lines. The first is the process id of the daemon and the second is the control file name of the current job being printed. The second line is updated to reflect the current status of *lpd* for the programs *lpq*(1) and *lprm*(1).

#### FILES

/etc/printcap	printer description file
/usr/spool/*	spool directories
/usr/spool/*/minfree	minimum free space to leave
/dev/lp*	line printer devices
/dev/printer	socket for local requests
/etc/hosts.equiv	lists machine names allowed printer access
/etc/hosts.lpd	lists machine names allowed printer access, but not under same administrative control.

#### SEE ALSO

*lpc*(8), *pac*(1), *lpr*(1), *lpq*(1), *lprm*(1), *syslog*(3), *printcap*(5)  
*4.2BSD Line Printer Spooler Manual*

**NAME**

makedev – make system special files

**SYNOPSIS**

/dev/MAKEDEV *device...*

**DESCRIPTION**

*MAKEDEV* is a shell script normally used to install special files. It resides in the */dev* directory, as this is the normal location of special files. Arguments to *MAKEDEV* are usually of the form *device-name?* where *device-name* is one of the supported devices listed in section 4 of the manual and “?” is a logical unit number (0-9). A few special arguments create assorted collections of devices and are listed below.

**std** Create the *standard* devices for the system; e.g. */dev/console*, */dev/tty*. The VAX-11/780 console floppy device, */dev/floppy*, and VAX-11/750 and VAX-11/730 console cassette device(s), */dev/tu?*, are also created with this entry.

**local** Create those devices specific to the local site. This request causes the shell file */dev/MAKEDEV.local* to be executed. Site specific commands, such as those used to setup dialup lines as “ttyd?” should be included in this file.

Since all devices are created using *mknod*(8), this shell script is useful only to the super-user.

**DIAGNOSTICS**

Either self-explanatory, or generated by one of the programs called from the script. Use “sh -x MAKEDEV” in case of trouble.

**SEE ALSO**

intro(4), config(8), mknod(8)

**NAME**

makekey – generate encryption key

**SYNOPSIS**

/usr/lib/makekey

**DESCRIPTION**

*Makekey* improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (that is, to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

*Makekey* is intended for programs that perform encryption (for instance, *ed* and *crypt(1)*). Usually *makekey*'s input and output will be pipes.

**SEE ALSO**

*crypt(1)*, *ed(1)*



**NAME**

**mkhosts** - generate hashed host table

**SYNOPSIS**

**/etc/mkhosts** [ -v ] *hostfile*

**DESCRIPTION**

*Mkhosts* is used to generate the hashed host database used by one version of the library routines `gethostbyaddr()` and `gethostbyname()`. It is not used if host name translation is performed by *named(8)*. If the **-v** option is supplied, each host will be listed as it is added. The file *hostfile* is usually */etc/hosts*, and in any case must be in the format of */etc/hosts* (see *hosts(5)*). *Mkhosts* will generate database files named *hostfile.pag* and *hostfile.dir*. The new database is built in a set of temporary files and only replaces the real database if the new one is built without errors. *Mkhosts* will exit with a non-zero exit code if any errors are detected.

**FILES**

*hostfile.pag*        - real database filenames  
*hostfile.dir*  
*hostfile.new.pag*   - temporary database filenames  
*hostfile.new.dir*

**SEE ALSO**

`gethostbyname(3)`, `gettable(8)`, `hosts(5)`, `htable(8)`, `named(8)`

**NAME**

mklost+found - make a lost+found directory for fsck

**SYNOPSIS**

/etc/mklost+found

**DESCRIPTION**

A directory *lost+found* is created in the current directory and a number of empty files are created therein and then removed so that there will be empty slots for *fsck*(8). This command should not normally be needed since *mkfs*(8) automatically creates the *lost+found* directory when a new file system is created.

**SEE ALSO**

fsck(8), mkfs(8)

**NAME**

mknod - build special file

**SYNOPSIS**

/etc/mknod name [ c ] [ b ] major minor

**DESCRIPTION**

*Mknod* makes a special file. The first argument is the *name* of the entry. The second is *b* if the special file is block-type (disks, tape) or *c* if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c*.

**SEE ALSO**

mknod(2), makedev(8)

**NAME**

mkpasswd - generate hashed password table

**SYNOPSIS**

/etc/mkpasswd [ -v ] passwdfile

**DESCRIPTION**

*Mkpasswd* is used to generate the hashed password database used by the library routines *getpwnam()* and *getpwuid()*. If the *-v* option is supplied, each entry will be listed as it is added. The file *passwdfile* is usually */etc/ptmp* (invoked by *vipw(8)*), and in any case must be in the format of */etc/passwd* (see *passwd(5)*). *Mkpasswd* will generate database files named *passwdfile.pag* and *passwdfile.dir*. *Mkpasswd* will exit with a non-zero exit code if any errors are detected.

**FILES**

*passwdfile.pag* - database filenames  
*passwdfile.dir*

**SEE ALSO**

*getpwent(3)*, *vipw(8)*, *passwd(5)*



**NAME**

mkproto – construct a prototype file system

**SYNOPSIS**

/etc/mkproto special proto

**DESCRIPTION**

*Mkproto* is used to bootstrap a new file system. First a new file system is created using *newfs*(8). *Mkproto* is then used to copy files from the old file system into the new file system according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first tokens comprise the specification for the root directory. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters *-bcd* specify regular, block special, character special and directory files respectively.) The second character of the type is either *u* or *-* to specify set-user-id mode or not. The third is *g* or *-* for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see *chmod*(1).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkproto* makes the entries *.* and *..* and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token *\$*.

A sample prototype specification follows:

```

d--777 3 1
usr  d--777 3 1
      sh  ---755 3 1 /bin/sh
      ken d--755 6 1
          $
      b0  b--644 3 1 0 0
      c0  c--644 3 1 0 0
          $
$

```

**SEE ALSO**

fs(5), *.dir*(5), fsck(8), newfs(8)

**BUGS**

There should be some way to specify links.

There should be some way to specify bad blocks.

Mkproto can only be run on virgin file systems. It should be possible to copy files into existent file systems.

**NAME**

**mount, umount** – mount and dismount file system

**SYNOPSIS**

*/etc/mount* [ *special name* [ *-r* ] ]

*/etc/mount* *-a*

*/etc/umount* *special*

*/etc/umount* *-a*

**DESCRIPTION**

*Mount* announces to the system that a removable file system is present on the device *special*. The file *name* must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional argument *-r* indicates that the file system is to be mounted read-only.

*Umount* announces to the system that the removable file system previously mounted on device *special* is to be removed.

If the *-a* option is present for either *mount* or *umount*, all of the file systems described in */etc/fstab* are attempted to be mounted or unmounted. In this case, *special* and *name* are taken from */etc/fstab*. The *special* file name from */etc/fstab* is the block special name.

These commands maintain a table of mounted devices in */etc/mtab*. If invoked without an argument, *mount* prints the table.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

**FILES**

<i>/etc/mtab</i>	mount table
<i>/etc/fstab</i>	file system table

**SEE ALSO**

*mount*(2), *mtab*(5), *fstab*(5)

**BUGS**

Mounting file systems full of garbage will crash the system.

Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

**NAME**

named – Internet domain name server

**SYNOPSIS**

named [ *-d debuglevel* ] [ *-p port#* ] [ *bootfile* ]

**DESCRIPTION**

*Named* is the Internet domain name server (see RFC883 for more details). Without any arguments, *named* will read the default boot file */etc/named.boot*, read any initial data and listen for queries.

Options are:

- d** Print debugging information. A number after the “d” determines the level of messages printed.
- p** Use a different port number. The default is the standard port number as listed in */etc/services*.

Any additional argument is taken as the name of the boot file. The boot file contains information about where the name server is to get its initial data. The following is a small example:

```

;
;      boot file for name server
;
; type      domain      source file or host
;
domain      berkeley.edu
primary      berkeley.edu  named.db
secondary    cc.berkeley.edu 10.2.0.78 128.32.0.10
cache        named.ca

```

The first line specifies that “berkeley.edu” is the domain for which the server is authoritative. The second line states that the file “named.db” contains authoritative data for the domain “berkeley.edu”. The file “named.db” contains data in the master file format described in RFC883 except that all domain names are relative to the origin; in this case, “berkeley.edu” (see below for a more detailed description). The second line specifies that all authoritative data under “cc.berkeley.edu” is to be transferred from the name server at 10.2.0.78. If the transfer fails it will try 128.32.0.10 and continue trying the address, up to 10, listed on this line. The secondary copy is also authoritative for the specified domain. The fourth line specifies data in “named.ca” is to be placed in the cache (i.e., well known data such as locations of root domain servers). The file “named.ca” is in the same format as “named.db”.

The master file consists of entries of the form:

```

$INCLUDE <filename>
$ORIGIN <domain>
<domain> <opt_ttl> <opt_class> <type> <resource_record_data>

```

where *domain* is “.” for root, “@” for the current origin, or a standard domain name. If *domain* is a standard domain name that does not end with “.”, the current origin is appended to the domain. Domain names ending with “.” are unmodified. The *opt\_ttl* field is an optional integer number for the time-to-live field. It defaults to zero. The *opt\_class* field is the object address type; currently only one type is supported, IN, for objects connected to the DARPA Internet. The *type* field is one of the following tokens; the data expected in the *resource\_record\_data* field is in parentheses.

A	a host address (dotted quad)
NS	an authoritative name server (domain)
MX	a mail exchanger (domain)
CNAME	the canonical name for an alias (domain)
SOA	marks the start of a zone of authority (5 numbers (see RFC883))
MB	a mailbox domain name (domain)
MG	a mail group member (domain)
MR	a mail rename domain name (domain)
NULL	a null resource record (no format or data)
WKS	a well know service description (not implemented yet)
PTR	a domain name pointer (domain)
HINFO	host information (cpu_type OS_type)
MINFO	mailbox or mail list information (request_domain error_domain)

**NOTES**

The following signals have the specified effect when sent to the server process using the *kill(1)* command.

SIGHUP	Causes server to read named.boot and reload database.
SIGINT	Dumps current data base and cache to /usr/tmp/named_dump.db
SIGUSR1	Turns on debugging; each SIGUSR1 increments debug level.
SIGUSR2	Turns off debugging completely.

**FILES**

/etc/named.boot	name server configuration boot file
/etc/named.pid	the process id
/usr/tmp/named.run	debug output
/usr/tmp/named_dump.db	dump of the name servers database

**SEE ALSO**

*kill(1)*, *gethostbyname(3N)*, *signal(3c)*, *resolver(3)*, *resolver(5)*, RFC882, RFC883, RFC973, RFC974, *Name Server Operations Guide for BIND*

**NAME**

**ncheck** - generate names from i-numbers

**SYNOPSIS**

*/etc/ncheck* [ -i numbers ] [ -a ] [ -s ] filesystems ...

**DESCRIPTION**

**N.B.:** For most normal file system maintenance, the function of *ncheck* is subsumed by *fsck*(8).

*Ncheck* with no options generates a pathname vs. i-number list of all files on every specified filesystem. Names of directory files are followed by '/.'. The -i option reduces the report to only those files whose i-numbers follow. The -a option allows printing of the names '.', and '..', which are ordinarily suppressed. The -s option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

The report is in no useful order, and probably should be sorted.

**SEE ALSO**

*sort*(1), *dcheck*(8), *fsck*(8), *icheck*(8)

**DIAGNOSTICS**

When the filesystem structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

**NAME**

**newfs** – construct a new file system

**SYNOPSIS**

*/etc/newfs* [ **-N** ] [ **-v** ] [ **-n** ] [ *mkfs-options* ] *special disk-type*

**DESCRIPTION**

*Newfs* is a “friendly” front-end to the *mkfs*(8) program. *Newfs* will look up the type of disk a file system is being created on in the disk description file */etc/disktab*, calculate the appropriate parameters to use in calling *mkfs*, then build the file system by forking *mkfs* and, if the file system is a root partition, install the necessary bootstrap programs in the initial 8 sectors of the device. The **-n** option prevents the bootstrap programs from being installed. The **-N** option causes the file system parameters to be printed out without actually creating the file system.

If the **-v** option is supplied, *newfs* will print out its actions, including the parameters passed to *mkfs*.

Options which may be used to override default parameters passed to *mkfs* are:

- s size**     The size of the file system in sectors.
- b block-size**     The block size of the file system in bytes.
- f frag-size**     The fragment size of the file system in bytes.
- t #tracks/cylinder**
- c #cylinders/group**     The number of cylinders per cylinder group in a file system. The default value used is 16.
- m free space %**     The percentage of space reserved from normal users; the minimum free space threshold. The default value used is 10%.
- o optimization preference (“space” or “time”)**     The file system can either be instructed to try to minimize the time spent allocating blocks, or to try to minimize the space fragmentation on the disk. If the value of minfree (see above) is less than 10%, the default is to optimize for space; if the value of minfree greater than or equal to 10%, the default is to optimize for time.
- r revolutions/minute**     The speed of the disk in revolutions per minute (normally 3600).
- S sector-size**     The size of a sector in bytes (almost never anything but 512).
- i number of bytes per inode**     This specifies the density of inodes in the file system. The default is to create an inode for each 2048 bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given.

**FILES**

<i>/etc/disktab</i>	for disk geometry and file system partition information
<i>/etc/mkfs</i>	to actually build the file system
<i>/usr/mdec</i>	for boot strapping programs

**SEE ALSO**

*disktab*(5), *fs*(5), *diskpart*(8), *fsck*(8), *format*(8), *mkfs*(8), *tunefs*(8)

M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems* 2, 3. pp 181-197, August 1984. (reprinted in the System Manager's Manual, SMM:14)

**BUGS**

Should figure out the type of the disk without the user's help.

**NAME**

**pac** - printer/plotter accounting information

**SYNOPSIS**

```
/etc/pac [ -Pprinter ] [ -pprice ] [ -s ] [ -r ] [ -c ] [ -m ] [ name ... ]
```

**DESCRIPTION**

*Pac* reads the printer/plotter accounting files, accumulating the number of pages (the usual case) or feet (for raster devices) of paper consumed by each user, and printing out how much each user consumed in pages or feet and dollars. If any *names* are specified, then statistics are only printed for those users; usually, statistics are printed for every user who has used any paper.

The **-P** flag causes accounting to be done for the named printer. Normally, accounting is done for the default printer (site dependent) or the value of the environment variable **PRINTER** is used.

The **-p** flag causes the value *price* to be used for the cost in dollars instead of the default value of 0.02 or the price specified in */etc/printcap*.

The **-c** flag causes the output to be sorted by cost; usually the output is sorted alphabetically by name.

The **-r** flag reverses the sorting order.

The **-s** flag causes the accounting information to be summarized on the summary accounting file; this summarization is necessary since on a busy system, the accounting file can grow by several lines per day.

The **-m** flag causes the host name to be ignored in the accounting file. This allows for a user on multiple machines to have all of his printing charges grouped together.

**FILES**

<i>/usr/adm/?acct</i>	raw accounting files
<i>/usr/adm/?_sum</i>	summary accounting files
<i>/etc/printcap</i>	printer capability data base

**SEE ALSO**

*printcap*(5)

**BUGS**

The relationship between the computed price and reality is as yet unknown.



**NAME**

`ping` – send ICMP ECHO\_REQUEST packets to network hosts

**SYNOPSIS**

```
/etc/ping [ -r ] [ -v ] host [ packetsize ] [ count ]
```

**DESCRIPTION**

The DARPA Internet is a large and complex aggregation of network hardware, connected together by gateways. Tracking a single-point hardware or software failure can often be difficult. *Ping* utilizes the ICMP protocol's mandatory ECHO\_REQUEST datagram to elicit an ICMP ECHO\_RESPONSE from a host or gateway. ECHO\_REQUEST datagrams ("pings") have an IP and ICMP header, followed by a struct `timeval`, and then an arbitrary number of "pad" bytes used to fill out the packet. Default datagram length is 64 bytes, but this may be changed using the command-line option. Other options are:

- `-r` Bypass the normal routing tables and send directly to a host on an attached network. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it (e.g., after the interface was dropped by *routed*(8C)).
- `-v` Verbose output. ICMP packets other than ECHO\_RESPONSE that are received are listed.

When using *ping* for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be "pinged". *Ping* sends one datagram per second, and prints one line of output for every ECHO\_RESPONSE returned. No output is produced if there is no response. If an optional *count* is given, only that number of requests is sent. Round-trip times and packet loss statistics are computed. When all responses have been received or the program times out (with a *count* specified), or if the program is terminated with a SIGINT, a brief summary is displayed.

This program is intended for use in network testing, measurement and management. It should be used primarily for manual fault isolation. Because of the load it could impose on the network, it is unwise to use *ping* during normal operations or from automated scripts.

**AUTHOR**

Mike Muuss

**SEE ALSO**

*netstat*(1), *ifconfig*(8C)

## NAME

pstat - print system facts

## SYNOPSIS

/etc/pstat -aixptuT [ suboptions ] [ system ] [ corefile ]

## DESCRIPTION

*Pstat* interprets the contents of certain system tables. If *corefile* is given, the tables are sought there, otherwise in */dev/kmem*. The required namelist is taken from */vmunix* unless *system* is specified. Options are

-a Under -p, describe all process slots rather than just active ones.

-i Print the inode table with these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

L locked

U update time (*fs(5)*) must be corrected

A access time must be corrected

M file system is mounted here

W wanted by another process (L flag is on)

T contains a text file

C changed time must be corrected

S shared lock applied

E exclusive lock applied

Z someone waiting for a lock

CNT Number of open file table entries for this inode.

DEV Major and minor device number of file system in which this inode resides.

RDC Reference count of shared locks on the inode.

WRC Reference count of exclusive locks on the inode (this may be > 1 if, for example, a file descriptor is inherited across a fork).

INO I-number within the device.

MODE Mode bits, see *chmod(2)*.

NLK Number of links to this inode.

UID User ID of owner.

SIZ/DEV

Number of bytes in an ordinary file, or major and minor device of special file.

-x Print the text table with these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

T *ptrace(2)* in effect

W text not yet written on swap device

L loading in progress

K locked

w wanted (L flag is on)

P resulted from demand-page-from-inode exec format (see *execve(2)*)

DADDR Disk address in swap, measured in multiples of 512 bytes.

CADDR Head of a linked list of loaded processes using this text segment.

RSS Size of resident text, measured in multiples of 512 bytes.

SIZE Size of text segment, measured in multiples of 512 bytes.

IPTR Core location of corresponding inode.

CNT	Number of processes using this text segment.
CCNT	Number of processes in core using this text segment.
FORW	Forward link in free list.
BACK	Backward link in free list.
-p	Print process table for active processes with these headings:
LOC	The core location of this table entry.
S	Run state encoded thus:
	0 no process
	1 waiting for some event
	3 runnable
	4 being created
	5 being terminated
	6 stopped (by signal or under trace)
F	Miscellaneous state variables, or'ed together (hexadecimal):
	0001 loaded
	0002 the scheduler process
	0004 locked for swap out
	0008 swapped out
	0010 traced
	0020 used in tracing
	0080 in page-wait
	0100 prevented from swapping during <i>fork(2)</i>
	0200 will restore old mask after taking signal
	0400 exiting
	0800 doing physical I/O ( <i>bio.c</i> )
	1000 process resulted from a <i>vfork(2)</i> which is not yet complete
	2000 another flag for <i>vfork(2)</i>
	4000 process has no virtual memory, as it is a parent in the context of <i>vfork(2)</i>
	8000 process is demand paging data pages from its text inode.
	10000 process using sequential VM patterns
	20000 process using random VM patterns
	100000 using old 4.1-compatible signal semantics
	200000 process needs profiling tick
	400000 process is scanning descriptors during select
	1000000 process page tables have changed
POIP	number of pages currently being pushed out from this process.
PRI	Scheduling priority, see <i>setpriority(2)</i> .
SIG	Signals received (signals 1-32 coded in bits 0-31),
UID	Real user ID.
SLP	Amount of time process has been blocked.
TIM	Time resident in seconds; times over 127 coded as 127.
CPU	Weighted integral of CPU time, for scheduler.
NI	Nice level, see <i>setpriority(2)</i> .
PGRP	Process number of root of process group.
PID	The process ID number.
PPID	The process ID of parent process.
ADDR	If in core, the page frame number of the first page of the 'u-area' of the process. If swapped out, the position in the swap area measured in multiples of 512 bytes.
RSS	Resident set size - the number of physical page frames allocated to this process.
SRSS	RSS at last swap (0 if never swapped).

**SIZE** Virtual size of process image (data+stack) in multiples of 512 bytes.  
**WCHAN** Wait channel number of a waiting process.  
**LINK** Link pointer in list of runnable processes.  
**TEXTP** If text is pure, pointer to location of text table entry.  
**-t** Print table for terminals with these headings:  
**RAW** Number of characters in raw input queue.  
**CAN** Number of characters in canonicalized input queue.  
**OUT** Number of characters in putput queue.  
**MODE** See *tty(4)*.  
**ADDR** Physical device address.  
**DEL** Number of delimiters (newlines) in canonicalized input queue.  
**COL** Calculated column position of terminal.  
**STATE** Miscellaneous state variables encoded thus:  
     **T** delay timeout in progress  
     **W** waiting for open to complete  
     **O** open  
     **F** outq has been flushed during DMA  
     **C** carrier is on  
     **B** busy doing output  
     **A** process is awaiting output  
     **X** open for exclusive use  
     **S** output stopped  
     **H** hangup on close  
**PGRP** Process group for which this is controlling terminal.  
**DISC** Line discipline; blank is old *tty* OTTYDISC or "new *tty*" for NTTYDISC or "net" for NETLDISC (see *bk(4)*).  
**-u** print information about a user process; the next argument is its address as given by *ps(1)*. The process must be in main memory, or the file used can be a core image and the address 0. Only the fields located in the first page cluster can be located successfully if the process is in main memory.  
**-f** Print the open file table with these headings:  
**LOC** The core location of this table entry.  
**TYPE** The type of object the file table entry points to.  
**FLG** Miscellaneous state variables encoded thus:  
     **R** open for reading  
     **W** open for writing  
     **A** open for appending  
     **S** shared lock present  
     **X** exclusive lock present  
     **I** signal pgrp when data ready  
**CNT** Number of processes that know this open file.  
**MSG** Number of messages outstanding for this file.  
**DATA** The location of the inode table entry or socket structure for this file.  
**OFFSET** The file offset (see *lseek(2)*).  
**-s** print information about swap space usage; the number of (1k byte) pages used and free is given as well as the number of used pages which belong to text images.  
**-T** prints the number of used and free slots in the several system tables and is useful for checking to see how full system tables have become if the system is under heavy load.

**FILES**

/vmunix     namelist  
/dev/kmem   default source of tables

**SEE ALSO**

iostat(1), ps(1), systat(1), vmstat(1), stat(2), fs(5),  
K. Thompson, *UNIX Implementation*

**BUGS**

It would be very useful if the system recorded "maximum occupancy" on the tables reported by -T; even more useful if these tables were dynamically allocated.

**NAME**

quot – summarize file system ownership

**SYNOPSIS**

/etc/quot [ option ] ... [ filesystem ]

**DESCRIPTION**

*Quot* prints the number of blocks in the named *filesystem* currently owned by each user. If no *filesystem* is named, a default name is assumed. The following options are available:

- n Cause the pipeline `ncheck filesystem | sort +0n | quot -n filesystem` to produce a list of all files and their owners.
- c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.
- f Print count of number of files as well as space owned by each user.

**FILES**

Default file system varies with system.  
/etc/passwd to get user names

**SEE ALSO**

ls(1), du(1)

**NAME**

quotacheck - file system quota consistency checker

**SYNOPSIS**

```
/etc/quotacheck [ -v ] [ -p ] filesystem...  
/etc/quotacheck [ -v ] [ -p ] -a
```

**DESCRIPTION**

*Quotacheck* examines each file system, builds a table of current disc usage, and compares this table against that stored in the disc quota file for the file system. If any inconsistencies are detected, both the quota file and the current system copy of the incorrect quotas are updated (the latter only occurs if an active file system is checked).

If the *-a* flag is supplied in place of any file system names, *quotacheck* will check all the file systems indicated in */etc/fstab* to be read-write with disc quotas.

Normally *quotacheck* reports only those quotas modified. If the *-v* option is supplied, *quotacheck* will indicate the calculated disc quotas for each user on a particular file system.

If the *-p* flag is supplied then parallel passes will be run on the filesystems required, using the pass numbers in */etc/fstab* in an identical fashion to *fsck*(8).

*Quotacheck* expects each file system to be checked to have a quota file named *quotas* in the root directory. If none is present, *quotacheck* will ignore the file system.

*Quotacheck* is normally run at boot time from the */etc/rc.local* file, see *rc*(8), before enabling disc quotas with *quotaon*(8).

*Quotacheck* accesses the raw device in calculating the actual disc usage for each user. Thus, the file systems checked should be quiescent while *quotacheck* is running.

**FILES**

*/etc/fstab*        default file systems

**SEE ALSO**

quota(2), setquota(2), quotaon(8), fsck(8)

**NAME**

quotaon, quotaoff - turn file system quotas on and off

**SYNOPSIS**

/etc/quotaon [ -v ] *filesystems...*

/etc/quotaon [ -v ] -a

/etc/quotaoff [ -v ] *filesystems...*

/etc/quotaoff [ -v ] -a

**DESCRIPTION**

*Quotaon* announces to the system that disc quotas should be enabled on one or more file systems. The file systems specified must have entries in */etc/fstab* and be mounted at the time. The file system quota files must be present in the root directory of the specified file system and be named *quotas*. The optional argument *-v* causes *quotaon* to print a message for each file system where quotas are turned on. If, instead of a list of file systems, a *-a* argument is given to *quotaon*, all file systems in */etc/fstab* marked read-write with quotas will have their quotas turned on. This is normally used at boot time to enable quotas.

*Quotaoff* announces to the system that file systems specified should have any disc quotas turned off. As above, the *-v* forces a verbose message for each file system affected; and the *-a* option forces all file systems in */etc/fstab* to have their quotas disabled.

These commands update the status field of devices located in */etc/mstab* to indicate when quotas are on or off for each file system.

**FILES**

/etc/mstab	mount table
/etc/fstab	file system table

**SEE ALSO**

setquota(2), mtab(5), fstab(5)



**NAME**

`rc` – command script for auto-reboot and daemons

**SYNOPSIS**

`/etc/rc`  
`/etc/rc.local`

**DESCRIPTION**

*Rc* is the command script which controls the automatic reboot and *rc.local* is the script holding commands which are pertinent only to a specific site.

When an automatic reboot is in progress, *rc* is invoked with the argument *autoboot* and runs a *fsck* with option `-p` to “preen” all the disks of minor inconsistencies resulting from the last system shutdown and to check for serious inconsistencies caused by hardware or software failure. If this auto-check and repair succeeds, then the second part of *rc* is run.

The second part of *rc*, which is run after a auto-reboot succeeds and also if *rc* is invoked when a single user shell terminates (see *init*(8)), starts all the daemons on the system, preserves editor files and clears the scratch directory `/tmp`. *Rc.local* is executed immediately before any other commands after a successful *fsck*. Normally, the first commands placed in the *rc.local* file define the machine's name, using *hostname*(1), and save any possible core image that might have been generated as a result of a system crash, *savecore*(8). The latter command is included in the *rc.local* file because the directory in which core dumps are saved is usually site specific.

**SEE ALSO**

*init*(8), *reboot*(8), *savecore*(8)

**BUGS**

**NAME**

**rdump** – file system dump across the network

**SYNOPSIS**

**/etc/rdump** [ *key* [ *argument* ... ] filesystem ]

**DESCRIPTION**

*Rdump* copies to magnetic tape all files changed after a certain date in the *filesystem*. The command is identical in operation to *dump*(8) except the *f* key should be specified and the file supplied should be of the form *machine:device*.

*Rdump* creates a remote server, */etc/rmt*, on the client machine to access the tape device.

**SEE ALSO**

*dump*(8), *rmt*(8C)

**DIAGNOSTICS**

Same as *dump*(8) with a few extra related to the network.

## NAME

reboot - UNIX bootstrapping procedures

## SYNOPSIS

/etc/reboot [ -n ] [ -q ]

## DESCRIPTION

UNIX is started by placing it in memory at location zero and transferring to the entry point. Since the system is not reenterable, it is necessary to read it in from disk or tape each time it is to be bootstrapped.

**Rebooting a running system.** When a UNIX is running and a reboot is desired, *shutdown(8)* is normally used. If there are no users then */etc/reboot* can be used. Reboot causes the disks to be synced and allows the system to perform other shutdown activities such as resynchronizing hardware time-of-day clocks. A multi-user reboot (as described below) is then initiated. This causes a system to be booted and an automatic disk check to be performed. If all this succeeds without incident, the system is then brought up for many users.

Options to reboot are:

- n option avoids the sync. It can be used if a disk or the processor is on fire.
- q reboots quickly and ungracefully, without shutting down running processes first.

*Reboot* normally logs the reboot using *syslog(8)* and places a shutdown record in the login accounting file */usr/adm/wtmp*. These actions are inhibited if the -n or -q options are present.

**Power fail and crash recovery.** Normally, the system will reboot itself at power-up or after crashes. Provided the auto-restart is enabled on the machine front panel, an automatic consistency check of the file systems will be performed, and unless this fails, the system will resume multi-user operations.

**Cold starts.** These are processor type dependent. On an 11/780, there are two floppy files for each disk controller, both of which cause boots from unit 0 of the root file system of a controller located on mba0 or uba0. One gives a single user shell, while the other invokes the multi-user automatic reboot. Thus these files are HPS and HPM for the single and multi-user boot from MASSBUS RP06/RM03/RM05 disks, UPS and UPM for UNIBUS storage module controller and disks such as the EMULEX SC-21 and AMPEX 9300 pair, or HKS and HKM for RK07 disks. There is also a script for booting from the default device, which is normally a copy of one of the standard multi-user boot scripts, but which may be modified to perform other actions or to boot from a different unit. The situation on the 8600 is similar, with scripts loaded from the console RL02.

Giving the command

```
>>>BOOT HPM
```

Would boot the system from (e.g.) an RP06 and run the automatic consistency check as described in *fsck(8)*. (Note that it may be necessary to type control-P and halt the processor to gain the attention of the LSI-11 before getting the >>> prompt.) The command

```
>>>BOOT ANY
```

invokes a version of the boot program in a way which allows you to specify any system as the system to be booted. It reads from the console a device specification (see below) followed immediately by a pathname.

The scripts may be modified for local configuration if necessary. The boot device type is set in register 10 as the device major number. The flags and minor device are placed in register 11. The register is used in four one-byte fields; from least to most significant, they are boot flags (as defined in *<sys/reboot.h>*), disk partition, drive unit, and adaptor number (UNIBUS

or MASSBUS as appropriate).

On an 11/750, the `reset` button will boot from the device selected by the front panel boot device switch. In systems with RK07's, position B normally selects the RK07 for boot. This will boot multi-user. To boot from RK07 with boot flags you may specify

```
>>>B/n DMA0
```

where, giving a *n* of 1 causes the boot program to ask for the name of the system to be bootstrapped, giving a *n* of 2 causes the boot program to come up single user, and a *n* of 3 causes both of these actions to occur. The "DM" specifies RK07, the "A" represents the adaptor number (UNIBUS or MASSBUS), and the "0" is the drive unit number. Other disk types which may be used are DB (MASSBUS), DD (TU58), and DU (UDA-50/RA disk). A non-zero disk partition can be used by adding (partition times 1000 hex) to *n*.

The 11/750 boot procedure uses the boot roms to load block 0 off of the specified device. The `/usr/mdcc` directory contains a number of bootstrap programs for the various disks which should be placed in a new pack automatically by `news(8)` when the "a" partition file system on the pack is created.

On any processor, the `boot` program finds the corresponding file on the given device (`vmunix` by default), loads that file into memory location zero, and starts the program at the entry address specified in the program header (after clearing off the high bit of the specified entry address).

The file specifications used with "BOOT ANY" or "B/3" are of the form:

```
device(unit,minor)
```

where *device* is the type of the device to be searched, *unit* is 8 \* the mba or uba number plus the unit number of the disk or tape, and *minor* is the disk partition or tape file number. Normal line editing characters can be used when typing the file specification. The following list of supported devices may vary from installation to installation:

hp	MASSBUS disk drive
up	UNIBUS storage module drive
ht	TE16,TU45,TU77 on MASSBUS
mt	TU78 on MASSBUS
hk	RK07 on UNIBUS
ra	storage module on a UDA50
rb	storage module on a 730 IDC
rl	RL02 on UNIBUS
tm	TM11 emulation tape drives on UNIBUS
ts	TS11 on UNIBUS
ut	UNIBUS TU45 emulator

For example, to boot from a file system which starts at cylinder 0 of unit 0 of a MASSBUS disk, type "hp(0,0)vmunix" to the boot prompt; "up(0,0)vmunix" would specify a UNIBUS drive, "hk(0,0)vmunix" would specify an RK07 disk drive, "ra(0,0)vmunix" would specify a UDA50 disk drive, and "rb(0,0)vmunix" would specify a disk on a 730 IDC. For tapes, the minor device number gives a file offset.

On an 11/750 with patchable control store, microcode patches will be installed by `boot` if the file `psc750.bin` exists in the root of the filesystem from which the system is booted.

In an emergency, the bootstrap methods described in the paper "Installing and Operating 4.3bsd" can be used to boot from a distribution tape.

#### FILES

<code>/vmunix</code>	system code
<code>/boot</code>	system bootstrap
<code>/usr/mdcc/xxboot</code>	sector-0 boot block for 750, xx is disk type

/usr/mdec/bootxx      second-stage boot for 750, xx is disk type  
/usr/mdec/installboot      program to install boot blocks on 750  
/pcs750.bin      microcode patch file on 750

**SEE ALSO**

arff(8V), crash(8V), fsck(8), halt(8), init(8), newfs(8), rc(8), shutdown(8), syslogd(8)

**NAME**

renice – alter priority of running processes

**SYNOPSIS**

```
/etc/renice priority [ [ -p ] pid ... ] [ [ -g ] pgrp ... ] [ [ -u ] user ... ]
```

**DESCRIPTION**

*Renice* alters the scheduling priority of one or more running processes. The *who* parameters are interpreted as process ID's, process group ID's, or user names. *Renice*'ing a process group causes all processes in the process group to have their scheduling priority altered. *Renice*'ing a user causes all processes owned by the user to have their scheduling priority altered. By default, the processes to be affected are specified by their process ID's. To force *who* parameters to be interpreted as process group ID's, a *-g* may be specified. To force the *who* parameters to be interpreted as user names, a *-u* may be given. Supplying *-p* will reset *who* interpretation to be (the default) process ID's. For example,

```
/etc/renice +1 987 -u daemon root -p 32
```

would change the priority of process ID's 987 and 32, and all processes owned by users daemon and root.

Users other than the super-user may only alter the priority of processes they own, and can only monotonically increase their "nice value" within the range 0 to PRIO\_MAX (20). (This prevents overriding administrative fiat.) The super-user may alter the priority of any process and set the priority to any value in the range PRIO\_MIN (-20) to PRIO\_MAX. Useful priorities are: 20 (the affected processes will run only when nothing else in the system wants to), 0 (the "base" scheduling priority), anything negative (to make things go very fast).

**FILES**

/etc/passwd      to map user names to user ID's

**SEE ALSO**

getpriority(2), setpriority(2)

**BUGS**

Non super-users can not increase scheduling priorities of their own processes, even if they were the ones that decreased the priorities in the first place.

**NAME**

**repquota** – summarize quotas for a file system

**SYNOPSIS**

**repquota** *filesystems...*

**DESCRIPTION**

*Repquota* prints a summary of the disc usage and quotas for the specified file systems. For each user the current number files and amount of space (in kilobytes) is printed, along with any quotas created with *edquota*(8).

Only the super-user may view quotas which are not their own.

**FILES**

*quotas* at the root of each file system with quotas  
*/etc/fstab* for file system names and locations

**SEE ALSO**

*quota*(1), *quota*(2), *quotacheck*(8), *quotaon*(8), *edquota*(8)

**DIAGNOSTICS**

Various messages about inaccessible files; self-explanatory.

**NAME**

restore – incremental file system restore

**SYNOPSIS**

/etc/restore key [ name ... ]

**DESCRIPTION**

*Restore* reads tapes dumped with the *dump*(8) command. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying the files that are to be restored. Unless the *h* key is specified (see below), the appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

- r** The tape is read and loaded into the current directory. This should not be done lightly; the *r* key should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape after a full level zero restore. Thus

```
/etc/newfs /dev/rrp0g eagle
/etc/mount /dev/rp0g /mnt
cd /mnt
restore r
```

is a typical sequence to restore a complete dump. Another *restore* can be done to get an incremental dump in on top of this. Note that *restore* leaves a file *restoresymtab* in the root directory to pass information between incremental restore passes. This file should be removed when the last incremental tape has been restored.

A *dump*(8) followed by a *newfs*(8) and a *restore* is used to change the size of a file system.

- R** *Restore* requests a particular tape of a multi volume set on which to restart a full restore (see the *r* key above). This allows *restore* to be interrupted and then restarted.
- x** The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, and the *h* key is not specified, the directory is recursively extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the *h* key has been specified.
- t** The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the *h* key has been specified. Note that the *t* key replaces the function of the old *dumpdir* program.
- i** This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, *restore* provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

**ls [arg]** – List the current or specified directory. Entries that are directories are appended with a “/”. Entries that have been marked for extraction are prepended with a “\*”. If the verbose key is set the inode number of each entry is also listed.

**cd arg** – Change the current working directory to the specified argument.

**pwd** – Print the full pathname of the current working directory.



**add** [arg] – The current directory or specified argument is added to the list of files to be extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the *h* key is specified on the command line). Files that are on the extraction list are prepended with a “\*” when they are listed by *ls*.

**delete** [arg] – The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the *h* key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

**extract** – All the files that are on the extraction list are extracted from the dump tape. *Restore* will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

**setmodes** – All the directories that have been added to the extraction list have their owner, modes, and times set; nothing is extracted from the tape. This is useful for cleaning up after a restore has been prematurely aborted.

**verbose** – The sense of the *v* key is toggled. When set, the verbose key causes the *ls* command to list the inode numbers of all entries. It also causes *restore* to print out information about each file as it is extracted.

**help** – List a summary of the available commands.

**quit** – Restore immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

- b** The next argument to *restore* is used as the block size of the tape (in kilobytes). If the *-b* option is not specified, *restore* tries to determine the tape block size dynamically.
- f** The next argument to *restore* is used as the name of the archive instead of */dev/rmt?*. If the name of the file is “-”, *restore* reads from standard input. Thus, *dump(8)* and *restore* can be used in a pipeline to dump and restore a file system with the command  

```
dump 0f - /usr | (cd /mnt; restore xf -)
```
- v** Normally *restore* does its work silently. The *v* (verbose) key causes it to type the name of each file it treats preceded by its file type.
- y** *Restore* will not ask whether it should abort the restore if gets a tape error. It will always try to skip over the bad tape block(s) and continue as best it can.
- m** *Restore* will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete path-name to the file.
- h** *Restore* extracts the actual directory, rather than the files that it references. This prevents hierarchical restoration of complete subtrees from the tape.
- s** The next argument to *restore* is a number which selects the file on a multi-file dump tape. File numbering starts at 1.

#### DIAGNOSTICS

Complaints about bad key characters.

Complaints if it gets a read error. If *y* has been specified, or the user responds “*y*”, *restore* will attempt to continue the restore.

If the dump extends over more than one tape, *restore* will ask the user to change tapes. If the *x* or *i* key has been specified, *restore* will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by *restore*. Most checks are self-explanatory or can “never happen”. Common errors are given below.

Converting to new file system format.

A dump tape created from the old file system has been loaded. It is automatically converted to the new file system format.

<filename>: not found on tape

The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file <inumber>, got <inumber>

A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low

When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high

When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring <filename>

Tape read error while skipping over inode <inumber>

Tape read error while trying to resynchronize

A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped <num> blocks

After a tape read error, *restore* may have to resynchronize itself. This message lists the number of blocks that were skipped over.

#### FILES

/dev/rmt?      the default tape drive  
 /tmp/rstdir\*   file containing directories on the tape.  
 /tmp/rstmode\*   owner, mode, and time stamps for directories.  
 ./restoresymtable   information passed between incremental restores.

#### SEE ALSO

restore(8C) dump(8), newfs(8), mount(8), mkfs(8)

#### BUGS

*Restore* can get confused when doing incremental restores from dump tapes that were made on active file systems.

A level zero dump must be done after a full restore. Because *restore* runs in user code, it has no control over inode allocation; thus a full restore must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged.

**NAME**

`rexecd` – remote execution server

**SYNOPSIS**

`/etc/rexecd`

**DESCRIPTION**

*Rexecd* is the server for the *rexec*(3X) routine. The server provides remote execution facilities with authentication based on user names and passwords.

*Rexecd* listens for service requests at the port indicated in the “exec” service specification; see *services*(5). When a service request is received the following protocol is initiated:

- 1) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.
- 2) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the stderr. A second connection is then created to the specified port on the client's machine.
- 3) A null terminated user name of at most 16 characters is retrieved on the initial socket.
- 4) A null terminated, unencrypted password of at most 16 characters is retrieved on the initial socket.
- 5) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.
- 6) *Rexecd* then validates the user as is done at login time and, if the authentication was successful, changes to the user's home directory, and establishes the user and group protections of the user. If any of these steps fail the connection is aborted with a diagnostic message returned.
- 7) A null byte is returned on the initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rexecd*.

**DIAGNOSTICS**

Except for the last one listed below, all diagnostic messages are returned on the initial socket, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 7 above upon successful completion of all the steps prior to the command execution).

“username too long”

The name is longer than 16 characters.

“password too long”

The password is longer than 16 characters.

“command too long ”

The command line passed exceeds the size of the argument list (as configured into the system).

“Login incorrect.”

No password file entry for the user name existed.

“Password incorrect.”

The wrong was password supplied.

“No remote directory.”

The *chdir* command to the home directory failed.

**"Try again."**

A *fork* by the server failed.

**"<shellname>: ..."**

The user's login shell could not be started. This message is returned on the connection associated with the `stderr`, and is not preceded by a flag byte.

**SEE ALSO**

`rexec(3X)`

**BUGS**

Indicating "Login incorrect" as opposed to "Password incorrect" is a security breach which allows people to probe a system for users with null passwords.

A facility to allow all data and password exchanges to be encrypted should be present.

**NAME**

*rlogind* – remote login server

**SYNOPSIS**

*/etc/rlogind* [ -d ]

**DESCRIPTION**

*Rlogind* is the server for the *rlogin*(1C) program. The server provides a remote login facility with authentication based on privileged port numbers from trusted hosts.

*Rlogind* listens for service requests at the port indicated in the “login” service specification; see *services*(5). When a service request is received the following protocol is initiated:

- 1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.
- 2) The server checks the client's source address and requests the corresponding host name (see *gethostbyaddr*(3N), *hosts*(5) and *named*(8)). If the hostname cannot be determined, the dot-notation representation of the host address is used.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty*(4)), and manipulates file descriptors so that the slave half of the pseudo terminal becomes the *stdin*, *stdout*, and *stderr* for a login process. The login process is an instance of the *login*(1) program, invoked with the -r option. The login process then proceeds with the authentication process as described in *rshd*(8C), but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseudo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. In normal operation, the packet protocol described in *pty*(4) is invoked to provide ^S/^Q type facilities and propagate interrupt signals to the remote programs. The login process propagates the client terminal's baud rate and terminal type, as found in the environment variable, “TERM”; see *environ*(7). The screen or window size of the terminal is requested from the client, and window size changes from the client are propagated to the pseudo terminal.

**DIAGNOSTICS**

All diagnostic messages are returned on the connection associated with the *stderr*, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

“Try again.”

A *fork* by the server failed.

“/bin/sh: ...”

The user's login shell could not be started.

**BUGS**

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an “open” environment.

A facility to allow all data exchanges to be encrypted should be present.

A more extensible protocol should be used.

**NAME**

*rmt* – remote magtape protocol module

**SYNOPSIS**

*/etc/rmt*

**DESCRIPTION**

*Rmt* is a program used by the remote dump and restore programs in manipulating a magnetic tape drive through an interprocess communication connection. *Rmt* is normally started up with an *rexec*(3X) or *rcmd*(3X) call.

The *rmt* program accepts requests specific to the manipulation of magnetic tapes, performs the commands, then responds with a status indication. All responses are in ASCII and in one of two forms. Successful commands have responses of

*A**number*\n

where *number* is an ASCII representation of a decimal number. Unsuccessful commands are responded to with

*E**error-number*\n*error-message*\n,

where *error-number* is one of the possible error numbers described in *intro*(2) and *error-message* is the corresponding error string as printed from a call to *perrot*(3). The protocol is comprised of the following commands (a space is present between each token).

- O device mode** Open the specified *device* using the indicated *mode*. *Device* is a full path-name and *mode* is an ASCII representation of a decimal number suitable for passing to *open*(2). If a device had already been opened, it is closed before a new open is performed.
- C device** Close the currently open device. The *device* specified is ignored.
- L whence offset** Perform an *lseek*(2) operation using the specified parameters. The response value is that returned from the *lseek* call.
- W count** Write data onto the open device. *Rmt* reads *count* bytes from the connection, aborting if a premature end-of-file is encountered. The response value is that returned from the *write*(2) call.
- R count** Read *count* bytes of data from the open device. If *count* exceeds the size of the data buffer (10 kilobytes), it is truncated to the data buffer size. *Rmt* then performs the requested *read*(2) and responds with *Acount-read*\n if the read was successful; otherwise an error in the standard format is returned. If the read was successful, the data read is then sent.
- I operation count** Perform a *MTIOCOP ioctl*(2) command using the specified parameters. The parameters are interpreted as the ASCII representations of the decimal values to place in the *mt\_op* and *mt\_count* fields of the structure used in the *ioctl* call. The return value is the *count* parameter when the operation is successful.
- S** Return the status of the open device, as obtained with a *MTIOCGET ioctl* call. If the operation was successful, an "ack" is sent with the size of the status buffer, then the status buffer is sent (in binary).

Any other command causes *rmt* to exit.

**DIAGNOSTICS**

All responses are of the form described above.

**SEE ALSO**

rcmd(3X), rexec(3X), mtio(4), rdump(8C), rrestore(8C)

**BUGS**

People tempted to use this for a remote file access protocol are discouraged.

## NAME

`route` – manually manipulate the routing tables

## SYNOPSIS

`/etc/route [ -f ] [ -n ] [ command args ]`

## DESCRIPTION

*Route* is a program used to manually manipulate the network routing tables. It normally is not needed, as the system routing table management daemon, *routed*(8C), should tend to this task.

*Route* accepts two commands: *add*, to add a route, and *delete*, to delete a route.

All commands have the following syntax:

`/etc/route command [ net | host ] destination gateway [ metric ]`

where *destination* is the destination host or network, *gateway* is the next-hop gateway to which packets should be addressed, and *metric* is a count indicating the number of hops to the *destination*. The metric is required for *add* commands; it must be zero if the destination is on a directly-attached network, and nonzero if the route utilizes one or more gateways. If adding a route with metric 0, the gateway given is the address of this host on the common network, indicating the interface to be used for transmission. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. The optional keywords *net* and *host* force the destination to be interpreted as a network or a host, respectively. Otherwise, if the *destination* has a "local address part" of INADDR\_ANY, or if the *destination* is the symbolic name of a network, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. If the route is to a destination connected via a gateway, the *metric* should be greater than 0. All symbolic names specified for a *destination* or *gateway* are looked up first as a host name using *gethostbyname*(3N). If this lookup fails, *getnetbyname*(3N) is then used to interpret the name as that of a network.

*Route* uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the super-user may modify the routing tables.

If the *-f* option is specified, *route* will "flush" the routing tables of all gateway entries. If this is used in conjunction with one of the commands described above, the tables are flushed prior to the command's application.

The *-n* option prevents attempts to print host and network names symbolically when reporting actions.

## DIAGNOSTICS

"add [ *host* | *network* ] %s: gateway %s flags %x"

The specified route is being added to the tables. The values printed are from the routing table entry supplied in the *ioctl* call. If the gateway address used was not the primary address of the gateway (the first one returned by *gethostbyname*), the gateway address is printed numerically as well as symbolically.

"delete [ *host* | *network* ] %s: gateway %s flags %x"

As above, but when deleting an entry.

"%s %s done"

When the *-f* flag is specified, each routing table entry deleted is indicated with a message of this form.

"Network is unreachable"

An attempt to add a route failed because the gateway listed was not on a directly-connected network. The next-hop gateway must be given.



**“not in table”**

A delete operation was attempted for an entry which wasn't present in the tables.

**“routing table overflow”**

An add operation was attempted, but the system was low on resources and was unable to allocate memory to create the new entry.

**SEE ALSO**

intro(4N), routed(8C), XNSrouted(8C)

**NAME**

*routed* – network routing daemon

**SYNOPSIS**

```
/etc/routed [ -d ] [ -g ] [ -s ] [ -q ] [ -t ] [ 'logfile' ]
```

**DESCRIPTION**

*Routed* is invoked at boot time to manage the network routing tables. The routing daemon uses a variant of the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries. It used a generalized protocol capable of use with multiple address types, but is currently used only for Internet routing within a cluster of networks.

In normal operation *routed* listens on the *udp*(4P) socket for the *route* service (see *services*(5)) for routing information packets. If the host is an internetwork router, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When *routed* is started, it uses the *SIOCGIFCONF ioctl* to find those directly connected interfaces configured into the system and marked “up” (the software loopback interface is ignored). If multiple interfaces are present, it is assumed that the host will forward packets between networks. *Routed* then transmits a *request* packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for *request* and *response* packets from other hosts.

When a *request* packet is received, *routed* formulates a reply based on the information maintained in its internal tables. The *response* packet generated contains a list of known routes, each marked with a “hop count” metric (a count of 16, or greater, is considered “infinite”). The metric associated with each route returned provides a metric *relative to the sender*.

*Response* packets received by *routed* are used to update the routing tables if one of the following conditions is satisfied:

- (1) No routing table entry exists for the destination network or host, and the metric indicates the destination is “reachable” (i.e. the hop count is not infinite).
- (2) The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.
- (3) The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.
- (4) The new route describes a shorter route to the destination than the one currently stored in the routing tables; the metric of the new route is compared against the one stored in the table to decide this.

When an update is applied, *routed* records the change in its internal tables and updates the kernel routing table. The change is reflected in the next *response* packet sent.

In addition to processing incoming packets, *routed* also periodically checks the routing table entries. If an entry has not been updated for 3 minutes, the entry's metric is set to infinity and marked for deletion. Deletions are delayed an additional 60 seconds to insure the invalidation is propagated throughout the local internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks. The response is sent to the broadcast address on nets capable of that function, to the destination address on point-to-point links, and to the router's own address on other networks. The normal routing tables are bypassed when sending gratuitous responses. The reception of responses on each network is used to determine that the network and interface are functioning correctly. If no response is received on an interface, another route may be chosen to route around the interface, or the route may be dropped if no alternative is available.

*Routed* supports several options:

- d Enable additional debugging information to be logged, such as bad packets received.
- g This flag is used on internetwork routers to offer a route to the "default" destination. This is typically used on a gateway to the Internet, or on a gateway that uses another routing protocol whose routes are not reported to other local routers.
- s Supplying this option forces *routed* to supply routing information whether it is acting as an internetwork router or not. This is the default if multiple network interfaces are present, or if a point-to-point link is in use.
- q This is the opposite of the -s option.
- t If the -t option is specified, all packets sent or received are printed on the standard output. In addition, *routed* will not divorce itself from the controlling terminal so that interrupts from the keyboard will kill the process.

Any other argument supplied is interpreted as the name of file in which *routed*'s actions should be logged. This log contains information about any changes to the routing tables and, if not tracing all packets, a history of recent messages sent and received which are related to the changed route.

In addition to the facilities described above, *routed* supports the notion of "distant" *passive* and *active* gateways. When *routed* is started up, it reads the file */etc/gateways* to find gateways which may not be located using only information from the *SIOGIFCONF ioctl*. Gateways specified in this manner should be marked passive if they are not expected to exchange routing information, while gateways marked active should be willing to exchange routing information (i.e. they should have a *routed* process running on the machine). Passive gateways are maintained in the routing tables forever and information regarding their existence is included in any routing information transmitted. Active gateways are treated equally to network interfaces. Routing information is distributed to the gateway and if no routing information is received for a period of the time, the associated route is deleted. External gateways are also passive, but are not placed in the kernel routing table nor are they included in routing updates. The function of external entries is to inform *routed* that another routing process will install such a route, and that alternate routes to that destination should not be installed. Such entries are only required when both routers may learn of routes to the same destination.

The */etc/gateways* is comprised of a series of lines, each in the following format:

```
< net | host > name1 gateway name2 metric value < passive | active | external >
```

The net or host keyword indicates if the route is to a network or specific host.

*Name1* is the name of the destination network or host. This may be a symbolic name located in */etc/networks* or */etc/hosts* (or, if started after *named(8)*, known to the name server), or an Internet address specified in "dot" notation; see *inet(3N)*.

*Name2* is the name or address of the gateway to which messages should be forwarded.

*Value* is a metric indicating the hop count to the destination host or network.

One of the keywords *passive*, *active* or *external* indicates if the gateway should be treated as *passive* or *active* (as described above), or whether the gateway is external to the scope of the *routed* protocol.

Internetwork routers that are directly attached to the Arpanet or Milnet should use the Exterior Gateway Protocol (EGP) to gather routing information rather than using a static routing table of passive gateways. EGP is required in order to provide routes for local networks to the rest of the Internet system. Sites needing assistance with such configurations should contact the Computer Systems Research Group at Berkeley.

**FILES**

/etc/gateways for distant gateways

**SEE ALSO**

"Internet Transport Protocols", X SIS 028112, Xerox System Integration Standard.  
udp(4P), XNSrouted(8C), htable(8)

**BUGS**

The kernel's routing tables may not correspond to those of *routed* when redirects change or add routes. The only remedy for this is to place the routing process in the kernel.

*Routed* should incorporate other routing protocols, such as Xerox NS (*XNSrouted*(8C)) and EGP. Using separate processes for each requires configuration options to avoid redundant or competing routes.

*Routed* should listen to intelligent interfaces, such as an IMP, and to error protocols, such as ICMP, to gather more information. It does not always detect unidirectional failures in network interfaces (e.g., when the output side fails).

**NAME**

**rrestore** – restore a file system dump across the network

**SYNOPSIS**

**/etc/rrestore** [ key [ name ... ]

**DESCRIPTION**

*Rrestore* obtains from magnetic tape files saved by a previous *dump*(8). The command is identical in operation to *restore*(8) except the *f* key should be specified and the file supplied should be of the form *machine:device*.

*Rrestore* creates a remote server, */etc/rmt*, on the client machine to access the tape device.

**SEE ALSO**

*restore*(8), *rmt*(8C)

**DIAGNOSTICS**

Same as *restore*(8) with a few extra related to the network.

**NAME**

*rshd* – remote shell server

**SYNOPSIS**

*/etc/rshd*

**DESCRIPTION**

*Rshd* is the server for the *rcmd*(3X) routine and, consequently, for the *rsh*(1C) program. The server provides remote execution facilities with authentication based on privileged port numbers from trusted hosts.

*Rshd* listens for service requests at the port indicated in the “cmd” service specification; see *services*(5). When a service request is received the following protocol is initiated:

- 1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.
- 2) The server reads characters from the socket up to a null (“\0”) byte. The resultant string is interpreted as an ASCII number, base 10.
- 3) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the *stderr*. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0-1023.
- 4) The server checks the client's source address and requests the corresponding host name (see *gethostbyaddr*(3N), *hosts*(5) and *named*(8)). If the hostname cannot be determined, the dot-notation representation of the host address is used.
- 5) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the client's machine.
- 6) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the server's machine.
- 7) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.
- 8) *Rshd* then validates the user according to the following steps. The local (server-end) user name is looked up in the password file and a *chdir* is performed to the user's home directory. If either the lookup or *chdir* fail, the connection is terminated. If the user is not the super-user, (user id 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered “equivalent”. If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the super-user, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the user on the client's machine. If this lookup fails, the connection is terminated.
- 9) A null byte is returned on the initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rshd*.

**DIAGNOSTICS**

Except for the last one listed below, all diagnostic messages are returned on the initial socket, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 9 above upon successful completion of all the steps prior to the execution of the login shell).

"locuser too long"

The name of the user on the client's machine is longer than 16 characters.

"remuser too long"

The name of the user on the remote machine is longer than 16 characters.

"command too long "

The command line passed exceeds the size of the argument list (as configured into the system).

"Login incorrect."

No password file entry for the user name existed.

"No remote directory."

The *chdir* command to the home directory failed.

"Permission denied."

The authentication procedure described above failed.

"Can't make pipe."

The pipe needed for the *stderr*, wasn't created.

"Try again."

A *fork* by the server failed.

"<shellname>: ..."

The user's login shell could not be started. This message is returned on the connection associated with the *stderr*, and is not preceded by a flag byte.

#### SEE ALSO

rsh(1C), rcmd(3X)

#### BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

A more extensible protocol should be used.

**NAME**

*rwhod* – system status server

**SYNOPSIS**

*/etc/rwhod*

**DESCRIPTION**

*Rwhod* is the server which maintains the database used by the *rwho*(1C) and *ruptime*(1C) programs. Its operation is predicated on the ability to *broadcast* messages on a network.

*Rwhod* operates as both a producer and consumer of status information. As a producer of information it periodically queries the state of the system and constructs status messages which are broadcast on a network. As a consumer of information, it listens for other *rwhod* servers' status messages, validating them, then recording them in a collection of files located in the directory */usr/spool/rwho*.

The server transmits and receives messages at the port indicated in the “*rwho*” service specification; see *services*(5). The messages sent and received, are of the form:

```
struct outmp {
    char    out_line[8];/* tty name */
    char    out_name[8];/* user id */
    long    out_time; /* time on */
};

struct whod {
    char    wd_vers;
    char    wd_type;
    char    wd_fill[2];
    int     wd_sendtime;
    int     wd_recvtime;
    char    wd_hostname[32];
    int     wd_loadav[3];
    int     wd_boottime;
    struct  whoent {
        struct outmp we_utmp;
        int    we_idle;
    } wd_we[1024 / sizeof(struct whoent)];
};
```

All fields are converted to network byte order prior to transmission. The load averages are as calculated by the *w*(1) program, and represent load averages over the 5, 10, and 15 minute intervals prior to a server's transmission; they are multiplied by 100 for representation in an integer. The host name included is that returned by the *gethostname*(2) system call, with any trailing domain name omitted. The array at the end of the message contains information about the users logged in to the sending machine. This information includes the contents of the *utmp*(5) entry for each non-idle terminal line and a value indicating the time in seconds since a character was last received on the terminal line.

Messages received by the *rwho* server are discarded unless they originated at an *rwho* server's port. In addition, if the host's name, as specified in the message, contains any unprintable ASCII characters, the message is discarded. Valid messages received by *rwhod* are placed in files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most recent message, in the format described above.

Status messages are generated approximately once every 3 minutes. *Rwhod* performs an *nlist*(3) on */vmunix* every 30 minutes to guard against the possibility that this file is not the system image currently operating.



**SEE ALSO**

      rwho(1C), ruptime(1C)

**BUGS**

There should be a way to relay status information between networks. Status information should be sent only upon request rather than continuously. People often interpret the server dying or network communication failures as a machine going down.

**NAME**

rxformat - format floppy disks

**SYNOPSIS**

/etc/rxformat [ -d ] special

**DESCRIPTION**

The *rxformat* program formats a diskette in the specified drive associated with the special device *special*. ( *Special* is normally /dev/rx0, for drive 0, or /dev/rx1, for drive 1.) By default, the diskette is formatted single density; a -d flag may be supplied to force double density formatting. Single density is compatible with the IBM 3740 standard (128 bytes/sector). In double density, each sector contains 256 bytes of data.

Before formatting a diskette *rxformat* prompts for verification if standard input is a tty (this allows a user to cleanly abort the operation; note that formatting a diskette will destroy any existing data). Formatting is done by the hardware. All sectors are zero-filled.

**DIAGNOSTICS**

'No such device' means that the drive is not ready, usually because no disk is in the drive or the drive door is open. Other error messages are selfexplanatory.

**FILES**

/dev/rx?

**SEE ALSO**

rx(4V)

**AUTHOR**

Helge Skrivervik

**BUGS**

A floppy may not be formatted if the header info on sector 1, track 0 has been damaged. Hence, it is not possible to format a completely degaussed disk. (This is actually a problem in the hardware.)

**NAME**

sa, accton – system accounting

**SYNOPSIS**

```
/etc/sa [ -abcdDfijkKlnrstuv ] [ -S savacctfile ] [ -U usracctfile ] [ file ]
/etc/accton [ file ]
```

**DESCRIPTION**

With an argument naming an existing *file*, *accton* causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

*Sa* reports on, cleans up, and generally maintains accounting files.

*Sa* is able to condense the information in */usr/adm/acct* into a summary file */usr/adm/savacct* which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system */usr/adm/acct* can grow by 100 blocks per day. The summary file is normally read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; */usr/adm/acct* is the default.

Output fields are labeled: “cpu” for the sum of user+system time (in minutes), “re” for real time (also in minutes), “k” for cpu-time averaged core usage (in 1k units), “avio” for average number of i/o operations per execution. With options fields labeled “tio” for total i/o operations, “k\*sec” for cpu storage integral (kilo-core seconds), “u” and “s” for user and system cpu time alone (both in minutes) will sometimes appear.

There are near a googol of options:

- a Print all command names, even those containing unprintable characters and those used only once. By default, those are placed under the name ‘\*\*\*other.’
- b Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.
- c Besides total user, system, and real time for each command print percentage of total time over all commands.
- d Sort by average number of disk i/o operations.
- D Print and sort by total number of disk i/o operations.
- f Force no interactive threshold compression with -v flag.
- i Don't read in summary file.
- j Instead of total minutes time for each category, give seconds per call.
- k Sort by cpu-time average memory usage.
- K Print and sort by cpu-storage integral.
- l Separate system and user time; normally they are combined.
- m Print number of processes and number of CPU minutes for each user.
- n Sort by number of calls.
- r Reverse order of sort.
- s Merge accounting file into summary file */usr/adm/savacct* when done.
- t For each command report ratio of real time to the sum of user and system times.

- u Superseding all other flags, print for each command in the accounting file the user ID and command name.
- v Followed by a number *n*, types the name of each command used *n* times or fewer. Await a reply from the terminal; if it begins with 'y', add the command to the category '\*\*junk\*\*.' This is used to strip out garbage.
- S The following filename is used as the command summary file instead of */usr/adm/savacct*.
- U The following filename is used instead of */usr/adm/usracct* to accumulate the per-user statistics printed by the *-m* option.

**FILES**

<i>/usr/adm/acct</i>	raw accounting
<i>/usr/adm/savacct</i>	summary
<i>/usr/adm/usracct</i>	per-user summary

**SEE ALSO**

ac(8), acct(2)

**BUGS**

The number of options to this program is absurd.

**NAME**

savecore – save a core dump of the operating system

**SYNOPSIS**

*/etc/savecore dirname [ system ]*

**DESCRIPTION**

*Savecore* is meant to be called near the end of the */etc/rc* file. Its function is to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log.

*Savecore* checks the core dump to be certain it corresponds with the current running unix. If it does it saves the core image in the file *dirname/vmcore.n* and its brother, the namelist, *dirname/vmunix.n*. The trailing ".n" in the pathnames is replaced by a number which grows every time *savecore* is run in that directory.

Before *savecore* writes out a core image, it reads a number from the file *dirname/minfree*. If the number of free kilobytes on the filesystem which contains *dirname* is less than the number obtained from the *minfree* file, the core dump is not saved. If the *minfree* file does not exist, *savecore* always writes out the core file (assuming that a core dump was taken).

*Savecore* also logs a reboot message using facility LOG\_AUTH (see *syslog(3)*) If the system crashed as a result of a panic, *savecore* logs the panic string too.

If the core dump was from a system other than */vmunix*, the name of that system must be supplied as *sysname*.

**FILES**

*/vmunix*                      current UNIX

**BUGS**

Can be fooled into thinking a core dump is the wrong size.

## NAME

sendmail – send mail over the internet

## SYNOPSIS

```
/usr/lib/sendmail [ flags ] [ address ... ]
```

```
newaliases
```

```
mailq [ -v ]
```

## DESCRIPTION

*Sendmail* sends a message to one or more *recipients*, routing the message over whatever networks are necessary. *Sendmail* does internetwork forwarding as necessary to deliver the message to the correct place.

*Sendmail* is not intended as a user interface routine; other programs provide user-friendly front ends; *sendmail* is used only to deliver pre-formatted messages.

With no flags, *sendmail* reads its standard input up to an end-of-file or a line consisting only of a single dot and sends a copy of the message found there to all of the addresses listed. It determines the network(s) to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'.

Flags are:

- ba** Go into ARPANET mode. All input lines must end with a CR-LF, and all messages will be generated with a CR-LF at the end. Also, the "From:" and "Sender:" fields are examined for the name of the sender.
- bd** Run as a daemon. This requires Berkeley IPC. *Sendmail* will fork and run in background listening on socket 25 for incoming SMTP connections. This is normally run from */etc/rc*.
- bi** Initialize the alias database.
- bm** Deliver mail in the usual way (default).
- bp** Print a listing of the queue.
- bs** Use the SMTP protocol as described in RFC821 on standard input and output. This flag implies all the operations of the **-ba** flag that are compatible with SMTP.
- bt** Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.
- bv** Verify names only – do not try to collect or deliver a message. Verify mode is normally used for validating users or mailing lists.
- bz** Create the configuration freeze file.
- Cfile** Use alternate configuration file. *Sendmail* refuses to run as root if an alternate configuration file is specified. The frozen configuration file is bypassed.
- dX** Set debugging value to *X*.
- Ffullname** Set the full name of the sender.
- fname** Sets the name of the "from" person (i.e., the sender of the mail). **-f** can only be used by "trusted" users (normally *root*, *daemon*, and *network*) or if the person you are trying to become is the same as the person you are.

- hN** Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop. If not specified, "Received:" lines in the message are counted.
  - n** Don't do aliasing.
  - ox value** Set option *x* to the specified *value*. Options are described below.
  - q[time]** Processed saved messages in the queue at given intervals. If *time* is omitted, process the queue once. *Time* is given as a tagged number, with 's' being seconds, 'm' being minutes, 'h' being hours, 'd' being days, and 'w' being weeks. For example, "-q1h30m" or "-q90m" would both set the timeout to one hour thirty minutes. If *time* is specified, *sendmail* will run in background. This option can be used safely with **-bd**.
  - rname** An alternate and obsolete form of the **-f** flag.
  - t** Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for recipient addresses. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed, that is, they will *not* receive copies even if listed in the message header.
  - v** Go into verbose mode. Alias expansions will be announced, etc.
- There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the command line using the **-o** flag or in the configuration file. These are described in detail in the *Sendmail Installation and Operation Guide*. The options are:
- Afile** Use alternate alias file.
  - c** On mailers that are considered "expensive" to connect to, don't initiate immediate connection. This requires queueing.
  - dx** Set the delivery mode to *x*. Delivery modes are 'i' for interactive (synchronous) delivery, 'b' for background (asynchronous) delivery, and 'q' for queue only - i.e., actual delivery is done the next time the queue is run.
  - D** Try to automatically rebuild the alias database if necessary.
  - ex** Set error processing to mode *x*. Valid modes are 'm' to mail back the error message, 'w' to "write" back the error message (or mail it back if the sender is not logged in), 'p' to print the errors on the terminal (default), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. If the text of the message is not mailed back by modes 'm' or 'w' and if the sender is local to this machine, a copy of the message is appended to the file "dead.letter" in the sender's home directory.
  - Fmode** The mode to use when creating temporary files.
  - f** Save UNIX-style From lines at the front of messages.
  - gN** The default group id to use when calling mailers.
  - Hfile** The SMTP help file.
  - i** Do not take dots on a line by themselves as a message terminator.
  - Ln** The log level.
  - m** Send to "me" (the sender) also if I am in an alias expansion.

<code>o</code>	If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.
<code>Qqueuedir</code>	Select the directory in which to queue messages.
<code>rtimeout</code>	The timeout on reads; if none is set, <i>sendmail</i> will wait forever for a mailer. This option violates the word (if not the intent) of the SMTP specification, show the timeout should probably be fairly large.
<code>Sfile</code>	Save statistics in the named file.
<code>s</code>	Always instantiate the queue file, even under circumstances where it is not strictly necessary. This provides safety against system crashes during delivery.
<code>Ttime</code>	Set the timeout on undelivered messages in the queue to the specified time. After delivery has failed (e.g., because of a host being down) for this amount of time, failed messages will be returned to the sender. The default is three days.
<code>tstz,dtz</code>	Set the name of the time zone.
<code>uN</code>	Set the default user id for mailers.

In aliases, the first character of a name may be a vertical bar to cause interpretation of the rest of the name as a command to pipe the mail to. It may be necessary to quote the name to keep *sendmail* from suppressing the blanks from between arguments. For example, a common alias is:

```
msgs: "|/usr/ucb/msgs -s"
```

Aliases may also have the syntax `":include:filename"` to ask *sendmail* to read the named file for a list of recipients. For example, an alias such as:

```
poets: ":include:/usr/local/lib/poets.list"
```

would read */usr/local/lib/poets.list* for the list of addresses making up the group.

*Sendmail* returns an exit status describing what it did. The codes are defined in `<sys/exits.h>`

<code>EX_OK</code>	Successful completion on all addresses.
<code>EX_NOUSER</code>	User name not recognized.
<code>EX_UNAVAILABLE</code>	Catchall meaning necessary resources were not available.
<code>EX_SYNTAX</code>	Syntax error in address.
<code>EX_SOFTWARE</code>	Internal software error, including bad arguments.
<code>EX_OSERR</code>	Temporary operating system error, such as "cannot fork".
<code>EX_NOHOST</code>	Host name not recognized.
<code>EX_TEMPFAIL</code>	Message could not be sent immediately, but was queued.

If invoked as *newaliases*, *sendmail* will rebuild the alias database. If invoked as *mailq*, *sendmail* will print the contents of the mail queue.

## FILES

Except for */usr/lib/sendmail.cf*, these pathnames are all specified in */usr/lib/sendmail.cf*. Thus, these values are only approximations.

<i>/usr/lib/aliases</i>	raw data for alias names
<i>/usr/lib/aliases.pag</i>	
<i>/usr/lib/aliases.dir</i>	data base of alias names
<i>/usr/lib/sendmail.cf</i>	configuration file
<i>/usr/lib/sendmail.fc</i>	frozen configuration
<i>/usr/lib/sendmail.hf</i>	help file



/usr/lib/sendmail.st	collected statistics
/usr/spool/mqueue/*	temp files

**SEE ALSO**

binmail(1), mail(1), rmail(1), syslog(3), aliases(5), sendmail.cf(5), mailaddr(7), rc(8);  
DARPA Internet Request For Comments RFC819, RFC821, RFC822;  
*Sendmail - An Internetwork Mail Router* (SMM:16);  
*Sendmail Installation and Operation Guide* (SMM:7)

**NAME**

shutdown – close down the system at a given time

**SYNOPSIS**

```
/etc/shutdown [ -k ] [ -r ] [ -h ] [ -f ] [ -n ] time [ warning-message ... ]
```

**DESCRIPTION**

*Shutdown* provides an automated shutdown procedure which a super-user can use to notify users nicely when the system is shutting down, saving them from system administrators, hackers, and gurus, who would otherwise not bother with niceties.

*Time* is the time at which *shutdown* will bring the system down and may be the word now (indicating an immediate shutdown) or specify a future time in one of two formats: +number and hour:min. The first form brings the system down in *number* minutes and the second brings the system down at the time of day indicated (as a 24-hour clock).

At intervals which get closer together as apocalypse approaches, warning messages are displayed at the terminals of all users on the system. Five minutes before shutdown, or immediately if shutdown is in less than 5 minutes, logins are disabled by creating */etc/nologin* and writing a message there. If this file exists when a user attempts to log in, *login(1)* prints its contents and exits. The file is removed just before *shutdown* exits.

At shutdown time a message is written in the system log, containing the time of shutdown, who ran shutdown and the reason. Then a terminate signal is sent to *init* to bring the system down to single-user state. Alternatively, if *-r*, *-h*, or *-k* was used, then *shutdown* will exec *reboot(8)*, *halt(8)*, or avoid shutting the system down (respectively). (If it isn't obvious, *-k* is to make people *think* the system is going down!)

With the *-f* option, *shutdown* arranges, in the manner of *fastboot(8)*, that when the system is rebooted the file systems will not be checked. The *-n* option prevents the normal *sync(2)* before stopping.

The time of the shutdown and the warning message are placed in */etc/nologin* and should be used to inform the users about when the system will be back up and why it is going down (or anything else).

**FILES**

*/etc/nologin*      tells login not to let anyone log in

**SEE ALSO**

*login(1)*, *reboot(8)*, *fastboot(8)*

**BUGS**

Only allows you to kill the system between now and 23:59 if you use the absolute time for shutdown.

**NAME**

**slattach** – attach serial lines as network interfaces

**SYNOPSIS**

*/etc/slattach* *ttynum* [ *baudrate* ]

**DESCRIPTION**

*Slattach* is used to assign a tty line to a network interface, and to define the network source and destination addresses. The *ttynum* parameter is a string of the form "ttyXX", or "/dev/ttyXX". The optional *baudrate* parameter is used to set the speed of the connection. If not specified, the default of 9600 is used.

Only the super-user may attach a network interface.

To detach the interface, use 'ifconfig *interface-name* down' after killing off the *slattach* process. *interface-name* is the name that is shown by netstat(1)

**EXAMPLES**

```
/etc/slattach ttyh8  
/etc/slattach /dev/tty01 4800
```

**DIAGNOSTICS**

Messages indicating the specified interface does not exist, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

**SEE ALSO**

rc(8), intro(4N), netstat(1), ifconfig(8C)

**NAME**

sticky – persistent text and append-only directories

**DESCRIPTION**

The *sticky bit* (file mode bit 01000, see *chmod(2)*) is used to indicate special treatment for certain executable files and directories.

**STICKY TEXT EXECUTABLE FILES**

While the 'sticky bit' is set on a sharable executable file, the text of that file will not be removed from the system swap area. Thus the file does not have to be fetched from the file system upon each execution. Shareable text segments are normally placed in a least-frequently-used cache after use, and thus the 'sticky bit' has little effect on commonly-used text images.

Sharable executable files are made by the *-n* and *-z* options of *ld(1)*.

Only the super-user can set the sticky bit on a sharable executable file.

**STICKY DIRECTORIES**

A directory whose 'sticky bit' is set becomes an append-only directory, or, more accurately, a directory in which the deletion of files is restricted. A file in a sticky directory may only be removed or renamed by a user if the user has write permission for the directory and the user is the owner of the file, the owner of the directory, or the super-user. This feature is usefully applied to directories such as */tmp* which must be publicly writable but should deny users the license to arbitrarily delete or rename each others' files.

Any user may create a sticky directory. See *chmod(1)* for details about modifying file modes.

**BUGS**

Since the text areas of sticky text executables are stashed in the swap area, abuse of the feature can cause a system to run out of swap.

Neither *open(2)* nor *mkdir(2)* will create a file with the sticky bit set.

**NAME**

swapon – specify additional device for paging and swapping

**SYNOPSIS**

*/etc/swapon* -a  
*/etc/swapon* name ...

**DESCRIPTION**

*Swapon* is used to specify additional devices on which paging and swapping are to take place. The system begins by swapping and paging on only a single device so that only one disk is required at bootstrap time. Calls to *swapon* normally occur in the system multi-user initialization file */etc/rc* making all swap devices available, so that the paging and swapping activity is interleaved across several devices.

Normally, the -a argument is given, causing all devices marked as “sw” swap devices in */etc/fstab* to be made available.

The second form gives individual block devices as given in the system swap configuration table. The call makes only this space available to the system for swap allocation.

**SEE ALSO**

swapon(2), init(8)

**FILES**

/dev/[ru][pk]?b normal paging devices

**BUGS**

There is no way to stop paging and swapping on a device. It is therefore not possible to make use of devices which may be dismounted during system operation.

**NAME**

`sync` – update the super block

**SYNOPSIS**

`/etc/sync`

**DESCRIPTION**

*Sync* executes the *sync* system primitive. *Sync* can be called to insure that all disk writes have been completed before the processor is halted in a way not suitably done by *reboot*(8) or *halt*(8). Generally, it is preferable to use *reboot* or *halt* to shut down the system, as they may perform additional actions such as resynchronizing the hardware clock and flushing internal caches before performing a final *sync*.

See *sync*(2) for details on the system primitive.

**SEE ALSO**

*sync*(2), *fsync*(2), *halt*(8), *reboot*(8), *update*(8)

**NAME**

syslogd – log systems messages

**SYNOPSIS**

```
/etc/syslogd [ -fconfigfile ] [ -mmarkinterval ] [ -d ]
```

**DESCRIPTION**

*Syslogd* reads and logs messages into a set of files described by the configuration file */etc/syslog.conf*. Each message is one line. A message can contain a priority code, marked by a number in angle braces at the beginning of the line. Priorities are defined in *<sys/syslog.h>*. *Syslogd* reads from the UNIX domain socket */dev/log*, from an Internet domain socket specified in */etc/services*, and from the special device */dev/klog* (to read kernel messages).

*Syslogd* configures when it starts up and whenever it receives a hangup signal. Lines in the configuration file have a *selector* to determine the message priorities to which the line applies and an *action*. The *action* field are separated from the selector by one or more tabs.

Selectors are semicolon separated lists of priority specifiers. Each priority has a *facility* describing the part of the system that generated the message, a dot, and a *level* indicating the severity of the message. Symbolic names may be used. An asterisk selects all facilities. All messages of the specified level or higher (greater severity) are selected. More than one facility may be selected using commas to separate them. For example:

```
*.emerg;mail,daemon.crit
```

Selects all facilities at the *emerg* level and the *mail* and *daemon* facilities at the *crit* level.

Known facilities and levels recognized by *syslogd* are those listed in *syslog(3)* without the leading "LOG\_". The additional facility "mark" has a message at priority LOG\_INFO sent to it every 20 minutes (this may be changed with the *-m* flag). The "mark" facility is not enabled by a facility field containing an asterisk. The level "none" may be used to disable a particular facility. For example,

```
*.debug;mail.none
```

Sends all messages *except* mail messages to the selected file.

The second part of each line describes where the message is to be logged if this line is selected. There are four forms:

- A filename (beginning with a leading slash). The file will be opened in append mode.
- A hostname preceded by an at sign ("@"). Selected messages are forwarded to the *syslogd* on the named host.
- A comma separated list of users. Selected messages are written to those users if they are logged in.
- An asterisk. Selected messages are written to all logged-in users.

Blank lines and lines beginning with '#' are ignored.

For example, the configuration file:

kern,mark.debug	/dev/console
*.notice;mail.info	/usr/spool/adm/syslog
*.crit	/usr/adm/critical
kern.err	@ucbarpa
*.emerg	*
*.alert	eric,kridle
*.alert;auth.warning	ralph

logs all kernel messages and 20 minute marks onto the system console, all notice (or higher) level messages and all mail system messages except debug messages into the file `/usr/spool/adm/syslog`, and all critical messages into `/usr/adm/critical`; kernel messages of error severity or higher are forwarded to `ucbarpa`. All users will be informed of any emergency messages, the users "eric" and "kridle" will be informed of any alert messages, and the user "ralph" will be informed of any alert message, or any warning message (or higher) from the authorization system.

The flags are:

- f Specify an alternate configuration file.
- m Select the number of minutes between mark messages.
- d Turn on debugging.

*Syslogd* creates the file `/etc/syslog.pid`, if possible, containing a single line with its process id. This can be used to kill or reconfigure *syslogd*.

To bring *syslogd* down, it should be sent a terminate signal (e.g. kill ``cat /etc/syslog.pid``).

#### FILES

<code>/etc/syslog.conf</code>	the configuration file
<code>/etc/syslog.pid</code>	the process id
<code>/dev/log</code>	Name of the UNIX domain datagram log socket
<code>/dev/klog</code>	The kernel log device

#### SEE ALSO

`logger(1)`, `syslog(3)`



**NAME**

talkd – remote user communication server

**SYNOPSIS**

/etc/talkd

**DESCRIPTION**

*Talkd* is the server that notifies a user that somebody else wants to initiate a conversation. It acts a repository of invitations, responding to requests by clients wishing to rendezvous to hold a conversation. In normal operation, a client, the caller, initiates a rendezvous by sending a CTL\_MSG to the server of type LOOK\_UP (see <protocols/talkd.h>). This causes the server to search its invitation tables to check if an invitation currently exists for the caller (to speak to the callee specified in the message). If the lookup fails, the caller then sends an ANNOUNCE message causing the server to broadcast an announcement on the callee's login ports requesting contact. When the callee responds, the local server uses the recorded invitation to respond with the appropriate rendezvous address and the caller and callee client programs establish a stream connection through which the conversation takes place.

**SEE ALSO**

talk(1), write(1)

**NAME**

**telnetd** – DARPA TELNET protocol server

**SYNOPSIS**

*/etc/telnetd*

**DESCRIPTION**

*Telnetd* is a server which supports the DARPA standard TELNET virtual terminal protocol. *Telnetd* is invoked by the internet server (see *inetd*(8)), normally for requests to connect to the TELNET port as indicated by the */etc/services* file (see *services*(5)).

*Telnetd* operates by allocating a pseudo-terminal device (see *pty*(4)) for a client, then creating a login process which has the slave side of the pseudo-terminal as *stdin*, *stdout*, and *stderr*. *Telnetd* manipulates the master side of the pseudo-terminal, implementing the TELNET protocol and passing characters between the remote client and the login process.

When a TELNET session is started up, *telnetd* sends TELNET options to the client side indicating a willingness to do *remote echo* of characters, to *suppress go ahead*, and to receive *terminal type information* from the remote client. If the remote client is willing, the remote terminal type is propagated in the environment of the created login process. The pseudo-terminal allocated to the client is configured to operate in “cooked” mode, and with XTABS and CRMOD enabled (see *tty*(4)).

*Telnetd* is willing to *do: echo, binary, suppress go ahead, and timing mark*. *Telnetd* is willing to have the remote client *do: binary, terminal type, and suppress go ahead*.

**SEE ALSO**

*telnet*(1C)

**BUGS**

Some TELNET commands are only partially implemented.

The TELNET protocol allows for the exchange of the number of lines and columns on the user's terminal, but *telnetd* doesn't make use of them.

Because of bugs in the original 4.2 BSD *telnet*(1C), *telnetd* performs some dubious protocol exchanges to try to discover if the remote client is, in fact, a 4.2 BSD *telnet*(1C).

*Binary mode* has no common interpretation except between similar operating systems (Unix in this case).

The terminal type name received from the remote client is converted to lower case.

The *packet* interface to the pseudo-terminal (see *pty*(4)) should be used for more intelligent flushing of input and output queues.

*Telnetd* never sends TELNET *go ahead* commands.

**NAME**

tftpd – DARPA Trivial File Transfer Protocol server

**SYNOPSIS**

/etc/tftpd

**DESCRIPTION**

*Tftpd* is a server which supports the DARPA Trivial File Transfer Protocol. The TFTP server operates at the port indicated in the “tftp” service description; see *services*(5). The server is normally started by *inetd*(8).

The use of *tftp* does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. Files may be written only if they already exist and are publicly writable. Note that this extends the concept of “public” to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling tftp service. The server should have the user ID with the lowest possible privilege.

**SEE ALSO**

tftp(1C), inetd(8)

**NAME**

`timed` - time server daemon

**SYNOPSIS**

`/etc/timed [ -t ] [ -M ] [ -n network ] [ -i network ]`

**DESCRIPTION**

*Timed* is the time server daemon and is normally invoked at boot time from the *rc*(8) file. It synchronizes the host's time with the time of other machines in a local area network running *timed*(8). These time servers will slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time. The average network time is computed from measurements of clock differences using the ICMP timestamp request message.

The service provided by *timed* is based on a master-slave scheme. When *timed*(8) is started on a machine, it asks the master for the network time and sets the host's clock to that time. After that, it accepts synchronization messages periodically sent by the master and calls *adjtime*(2) to perform the needed corrections on the host's clock.

It also communicates with *date*(1) in order to set the date globally, and with *timedc*(8), a timed control program. If the machine running the master crashes, then the slaves will elect a new master from among slaves running with the *-M* flag. A *timed* running without the *-M* flag will remain a slave. The *-t* flag enables *timed* to trace the messages it receives in the file */usr/adm/timed.log*. Tracing can be turned on or off by the program *timedc*(8). *Timed* normally checks for a master time server on each network to which it is connected, except as modified by the options described below. It will request synchronization service from the first master server located. If permitted by the *-M* flag, it will provide synchronization service on any attached networks on which no current master server was detected. Such a server propagates the time computed by the top-level master. The *-n* flag, followed by the name of a network which the host is connected to (see *networks*(5)), overrides the default choice of the network addresses made by the program. Each time the *-n* flag appears, that network name is added to a list of valid networks. All other networks are ignored. The *-i* flag, followed by the name of a network to which the host is connected (see *networks*(5)), overrides the default choice of the network addresses made by the program. Each time the *-i* flag appears, that network name is added to a list of networks to ignore. All other networks are used by the time daemon. The *-n* and *-i* flags are meaningless if used together.

**FILES**

<i>/usr/adm/timed.log</i>	tracing file for <i>timed</i>
<i>/usr/adm/timed.masterlog</i>	log file for master <i>timed</i>

**SEE ALSO**

*date*(1), *adjtime*(2), *gettimeofday*(2), *icmp*(4P), *timedc*(8),  
*TSP: The Time Synchronization Protocol for UNIX 4.3BSD*, R. Gusella and S. Zatti

**NAME**

timedc - timed control program

**SYNOPSIS**

/etc/timedc [ command [ argument ... ] ]

**DESCRIPTION**

*Timedc* is used to control the operation of the *timed* program. It may be used to:

- measure the differences between machines' clocks,
- find the location where the master time server is running,
- enable or disable tracing of messages received by *timed*, and
- perform various debugging actions.

Without any arguments, *timedc* will prompt for commands from the standard input. If arguments are supplied, *timedc* interprets the first argument as a command and the remaining arguments as parameters to the command. The standard input may be redirected causing *timedc* to read commands from a file. Commands may be abbreviated; recognized commands are:

? [ command ... ]

help [ command ... ]

Print a short description of each command specified in the argument list, or, if no arguments are given, a list of the recognized commands.

clockdiff host ...

Compute the differences between the clock of the host machine and the clocks of the machines given as arguments.

trace { on | off }

Enable or disable the tracing of incoming messages to *timed* in the file /usr/adm/timed.log.

quit

Exit from *timedc*.

Other commands may be included for use in testing and debugging *timed*; the help command and the program source may be consulted for details.

**FILES**

/usr/adm/timed.log tracing file for *timed*

/usr/adm/timed.masterloglog file for master *timed*

**SEE ALSO**

date(1), adjtime(2), icmp(4P), timed(8),

*TSP: The Time Synchronization Protocol for UNIX 4.3BSD*, R. Gusella and S. Zatti

**DIAGNOSTICS**

?Ambiguous command	abbreviation matches more than one command
?Invalid command	no match found
?Privileged command	command can be executed by root only

**NAME**

*trpt* – transliterate protocol trace

**SYNOPSIS**

*trpt* [ *-a* ] [ *-s* ] [ *-t* ] [ *-f* ] [ *-j* ] [ *-p* hex-address ] [ system [ core ] ]

**DESCRIPTION**

*Trpt* interrogates the buffer of TCP trace records created when a socket is marked for “debugging” (see *setsockopt(2)*), and prints a readable description of these records. When no options are supplied, *trpt* prints all the trace records found in the system grouped according to TCP connection protocol control block (PCB). The following options may be used to alter this behavior.

- a* in addition to the normal output, print the values of the source and destination addresses for each packet recorded.
- s* in addition to the normal output, print a detailed description of the packet sequencing information.
- t* in addition to the normal output, print the values for all timers at each point in the trace.
- f* follow the trace as it occurs, waiting a short time for additional records each time the end of the log is reached.
- j* just give a list of the protocol control block addresses for which there are trace records.
- p* show only trace records associated with the protocol control block, the address of which follows.

The recommended use of *trpt* is as follows. Isolate the problem and enable debugging on the socket(s) involved in the connection. Find the address of the protocol control blocks associated with the sockets using the *-A* option to *netstat(1)*. Then run *trpt* with the *-p* option, supplying the associated protocol control block addresses. The *-f* option can be used to follow the trace log once the trace is located. If there are many sockets using the debugging option, the *-j* option may be useful in checking to see if any trace records are present for the socket in question. The

If debugging is being performed on a system or core file other than the default, the last two arguments may be used to supplant the defaults.

**FILES**

/vmunix  
/dev/kmem

**SEE ALSO**

*setsockopt(2)*, *netstat(1)*, *trsp(8C)*

**DIAGNOSTICS**

“no namelist” when the system image doesn’t contain the proper symbols to find the trace buffer; others which should be self explanatory.

**BUGS**

Should also print the data for each input or output, but this is not saved in the race record. The output format is inscrutable and should be described here.

**NAME**

*trsp* - transliterate sequenced packet protocol trace

**SYNOPSIS**

*trsp* [ -a ] [ -s ] [ -t ] [ -j ] [ -p hex-address ] [ system [ core ] ]

**DESCRIPTION**

*Trpt* interrogates the buffer of SPP trace records created when a socket is marked for "debugging" (see *setsockopt(2)*), and prints a readable description of these records. When no options are supplied, *trsp* prints all the trace records found in the system grouped according to SPP connection protocol control block (PCB). The following options may be used to alter this behavior.

- s in addition to the normal output, print a detailed description of the packet sequencing information,
- t in addition to the normal output, print the values for all timers at each point in the trace,
- j just give a list of the protocol control block addresses for which there are trace records,
- p show only trace records associated with the protocol control block whose address follows,
- a in addition to the normal output, print the values of the source and destination addresses for each packet recorded.

The recommended use of *trsp* is as follows. Isolate the problem and enable debugging on the socket(s) involved in the connection. Find the address of the protocol control blocks associated with the sockets using the -A option to *netstat(1)*. Then run *trsp* with the -p option, supplying the associated protocol control block addresses. If there are many sockets using the debugging option, the -j option may be useful in checking to see if any trace records are present for the socket in question.

If debugging is being performed on a system or core file other than the default, the last two arguments may be used to supplant the defaults.

**FILES**

/vmunix  
/dev/kmem

**SEE ALSO**

*setsockopt(2)*, *netstat(1)*

**DIAGNOSTICS**

"no namelist" when the system image doesn't contain the proper symbols to find the trace buffer; others which should be self explanatory.

**BUGS**

Should also print the data for each input or output, but this is not saved in the trace record. The output format is inscrutable and should be described here.

**NAME**

tunefs - tune up an existing file system

**SYNOPSIS**

/etc/tunefs tuneup-options special/filesys

**DESCRIPTION**

*Tunefs* is designed to change the dynamic parameters of a file system which affect the layout policies. The parameters which are to be changed are indicated by the flags given below:

**-a maxcontig**

This specifies the maximum number of contiguous blocks that will be laid out before forcing a rotational delay (see **-d** below). The default value is one, since most device drivers require an interrupt per disk transfer. Device drivers that can chain several buffers together in a single transfer should set this to the maximum chain length.

**-d rotdelay**

This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file.

**-e maxbpg**

This indicates the maximum number of blocks any single file can allocate out of a cylinder group before it is forced to begin allocating blocks from another cylinder group. Typically this value is set to about one quarter of the total blocks in a cylinder group. The intent is to prevent any single file from using up all the blocks in a single cylinder group, thus degrading access times for all files subsequently allocated in that cylinder group. The effect of this limit is to cause big files to do long seeks more frequently than if they were allowed to allocate all the blocks in a cylinder group before seeking elsewhere. For file systems with exclusively large files, this parameter should be set higher.

**-m minfree**

This value specifies the percentage of space held back from normal users; the minimum free space threshold. The default value used is 10%. This value can be set to zero, however up to a factor of three in throughput will be lost over the performance obtained at a 10% threshold. Note that if the value is raised above the current usage level, users will be unable to allocate files until enough files have been deleted to get under the higher threshold.

**-o optimization preference**

The file system can either try to minimize the time spent allocating blocks, or it can attempt minimize the space fragmentation on the disk. If the value of minfree (see above) is less than 10%, then the file system should optimize for space to avoid running out of full sized blocks. For values of minfree greater than or equal to 10%, fragmentation is unlikely to be problematical, and the file system can be optimized for time.

**SEE ALSO**

fs(5), newfs(8), mkfs(8)

M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems* 2, 3. pp 181-197, August 1984. (reprinted in the System Manager's Manual, SMM:14)

**BUGS**

This program should work on mounted and active file systems. Because the super-block is not kept in the buffer cache, the changes will only take effect if the program is run on dismounted file systems. To change the root file system, the system must be rebooted after



the file system is tuned.

You can tune a file system, but you can't tune a fish.

**NAME**

**update** – periodically update the super block

**SYNOPSIS**

*/etc/update*

**DESCRIPTION**

*Update* is a program that executes the *sync(2)* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

**SEE ALSO**

*sync(2)*, *sync(8)*, *init(8)*, *rc(8)*

**BUGS**

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync(8)* temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.

**NAME**

*uucico*, *uucpd* – transfer files queued by *uucp* or *uux*

**SYNOPSIS**

```
/usr/lib/uucp/uucico [ -dspooldir ] [ -ggrade ] [ -rrole ] [ -R ] [ -ssystem ] [ -xdebug ] [ -L ] [
-tturnaround ]
/etc/uucpd
```

**DESCRIPTION**

*Uucico* performs the actual work involved in transferring files between systems. *Uucp*(1C) and *uux*(1C) merely queue requests for data transfer which *uucico* processes.

The following options are available.

**-dspooldir**

Use *spooldir* as the spool directory. The default is */usr/spool/uucp*.

**-ggrade** Only send jobs of grade *grade* or higher this transfer. The grade of a job is specified when the job is queued by *uucp* or *uux*.

**-rrole** *role* is either 1 or 0; it indicates whether *uucico* is to start up in master or slave role, respectively. 1 is used when running *uucico* by hand or from *cron*(8). 0 is used when another system calls the local system. Slave role is the default.

**-R** Reverse roles. When used with the *-r1* option, this tells the remote system to begin sending its jobs first, instead of waiting for the local machine to finish.

**-ssystem**

Call only system *system*. If *-s* is not specified, and *-r1* is specified, *uucico* will attempt to call all systems for which there is work. If *-s* is specified, a call will be made even if there is no work for that system. This is useful for polling.

**-xdebug**

Turn on debugging at level *debug*. Level 5 is a good start when trying to find out why a call failed. Level 9 is very detailed. Level 99 is absurdly verbose. If *role* is 1 (master), output is normally written to the standard message output *stderr*. If *stderr* is unavailable, output is written to */usr/spool/uucp/AUDIT/system*. When *role* is 0 (slave), debugging output is always written to the AUDIT file.

**-L** Only call "local" sites. A site is considered local if the device-type field in *L.sys* is one of LOCAL, DIR or TCP.

**-tturnaround**

Use *turnaround* as the line turnaround time (in minutes) instead of the default 30. If *turnaround* is missing or 0, line turnaround will be disabled. After *uucico* has been running in slave role for *turnaround* minutes, it will attempt to run in master role by negotiating with the remote machine. In earlier versions of *uucico*, a transfer of many large files in one direction would hold up mail going in the other direction. With the turnaround code working, the message flow will be more bidirectional in the short term. This option only works with newer *uucico*'s and is ignored by older ones.

If *uucico* receives a SIGFPE (see *kill*(1)), it will toggle the debugging on or off.

*Uucpd* is the server for supporting *uucp* connections over networks. *Uucpd* listens for service requests at the port indicated in the "uucp" service specification; see *services*(5). The server provides login name and password authentication before starting up *uucico* for the rest of the transaction.

*Uucico* is commonly used either of two ways: as a daemon run periodically by *cron*(8) to call out to remote systems, and as a "shell" for remote systems who call in. For calling out

periodically, a typical line in *crontab* would be:

```
0 * * * * /usr/lib/uucp/uucico -r1
```

This will run *uucico* every hour in master role. For each system that has transfer requests queued, *uucico* calls the system, logs in, and executes the transfers. The file *L.sys*(5) is consulted for information about how to log in, while *L-devices*(5) specifies available lines and modems for calling.

For remote systems to dial in, an entry in the *passwd*(5) file must be created, with a login "shell" of *uucico*. For example:

```
uucp:Password:6:1::/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

The UID for UUCP remote logins is not critical, so long as it differs from the UUCP Administrative login. The latter owns the UUCP files, and assigning this UID to a remote login would be an extreme security hazard.

#### FILES

/usr/lib/uucp/	UUCP internal files/utilities
/usr/lib/uucp/L-devices	Local device descriptions
/usr/lib/uucp/L-dialcodes	Phone numbers and prefixes
/usr/lib/uucp/L.aliaes	Hostname aliases
/usr/lib/uucp/L.cmds	Remote command permissions list
/usr/lib/uucp/L.sys	Host connection specifications
/usr/lib/uucp/USERFILE	Remote directory tree permissions list
/usr/spool/uucp/	Spool directory
/usr/spool/uucp/AUDIT/*	Debugging audit trails
/usr/spool/uucp/C./	Control files directory
/usr/spool/uucp/D./	Incoming data file directory
/usr/spool/uucp/D.hostname/	Outgoing data file directory
/usr/spool/uucp/D.hostnameX/	Outgoing execution file directory
/usr/spool/uucp/CORRUPT/	Place for corrupted C. and D. files
/usr/spool/uucp/ERRLOG	UUCP internal error log
/usr/spool/uucp/LOGFILE	UUCP system activity log
/usr/spool/uucp/LCK/LCK.*	Device lock files
/usr/spool/uucp/SYSLOG	File transfer statistics log
/usr/spool/uucp/STST/*	System status files
/usr/spool/uucp/TM./	File transfer temp directory
/usr/spool/uucp/X./	Incoming execution file directory
/usr/spool/uucppublic	Public access directory

#### SEE ALSO

*uucp*(1C), *uuc*(1C), *uux*(1C), *L-devices*(5), *L-dialcodes*(5), *L.aliaes*(5), *L.cmds*(5), *L.sys*(5), *uuclean*(8C), *uupoll*(8C), *uusnap*(8C), *uuxqt*(8C)

D. A. Nowitz and M. E. Lesk, *A Dial-Up Network of UNIX Systems*.

D. A. Nowitz, *Uucp Implementation Description*.

**NAME**

uuclean - uucp spool directory clean-up

**SYNOPSIS**

/usr/lib/uucp/uuclean [ -m ] [ -ntime ] [ -ppre ]

**DESCRIPTION**

*Uuclean* will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

The following options are available.

- ppre Scan for files with *pre* as the file prefix. Up to 10 -p arguments may be specified.
- ntime Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)
- m Send mail to the owner of the file when it is deleted.
- dsubdirectory Only the specified subdirectory will be cleaned.

This program will typically be run daily by *cron*(8).

**FILES**

/usr/spool/uucp           Spool directory

**SEE ALSO**

uucp(1C), uux(1C), uucico(8C)

**NAME**

**uupoll** – poll a remote UUCP site

**SYNOPSIS**

**uupoll** [ **-ggrade** ] [ **-n** ] *system*

**DESCRIPTION**

*Uupoll* is used to force a poll of a remote system. It queues a null job for the remote system and then invokes *uucico*(8C).

The following options are available:

**-ggrade** Only send jobs of grade *grade* or higher on this call.

**-n** Queue the null job, but do not invoke *uucico*.

*Uupoll* is usually run by *cron*(5) or by a user who wants to hurry a job along. A typical entry in *crontab* could be:

```
0    0,8,16 * * * /usr/bin/uupoll ihnp4
0    4,12,20 * * * /usr/bin/uupoll ucbvax
```

This will poll *ihnp4* at midnight, 0800, and 1600, and *ucbvax* at 0400, noon, and 2000.

If the local machine is already running *uucico* every hour and has a limited number of outgoing modems, a more elegant approach might be:

```
0    0,8,16 * * * /usr/bin/uupoll -n ihnp4
0    4,12,20 * * * /usr/bin/uupoll -n ucbvax
5    * * * * * /usr/lib/uucp/uucico -r1
```

This will queue null jobs for the remote sites at the top of hour; they will be processed by *uucico* when it runs five minutes later.

**FILES**

*/usr/lib/uucp/* UUCP internal files/utilities  
*/usr/spool/uucp/* Spool directory

**SEE ALSO**

*uucp*(1C), *uux*(1C), *uucico*(8C)

**NAME**

uusnap – show snapshot of the UUCP system

**SYNOPSIS**

uusnap

**DESCRIPTION**

*Uusnap* displays in tabular format a synopsis of the current UUCP situation. The format of each line is as follows:

site	N Cmds	N Data	N Xqts	Message
------	--------	--------	--------	---------

Where "site" is the name of the site with work, "N" is a count of each of the three possible types of work (command, data, or remote execute), and "Message" is the current status message for that site as found in the STST file.

Included in "Message" may be the time left before UUCP can re-try the call, and the count of the number of times that UUCP has tried (unsuccessfully) to reach the site.

**SEE ALSO**

uucp(1C), uux(1C), uuq(1C), uucico(8C)  
*UUCP Implementation Guide*

**NAME**

*uuxqt* - UUCP execution file interpreter

**SYNOPSIS**

*/usr/lib/uucp/uuxqt* [ *-xdebug* ]

**DESCRIPTION**

*Uuxqt* interprets *execution files* created on a remote system via *uux*(1C) and transferred to the local system via *uucico*(8C). When a user uses *uux* to request remote command execution, it is *uuxqt* that actually executes the command. Normally, *uuxqt* is forked from *uucico* to process queued execution files; for debugging, it may also be run manually by the UUCP administrator.

*Uuxqt* runs in its own subdirectory, */usr/spool/uucp/XTMP*. It copies intermediate files to this directory when necessary.

**FILES**

<i>/usr/lib/uucp/L.cmds</i>	Remote command permissions list
<i>/usr/lib/uucp/USERFILE</i>	Remote directory tree permissions list
<i>/usr/spool/uucp/LOGFILE</i>	UUCP system activity log
<i>/usr/spool/uucp/LCK/LCK.XQT</i>	<i>Uuxqt</i> lock file
<i>/usr/spool/uucp/X./</i>	Incoming execution file directory
<i>/usr/spool/uucp/XTMP</i>	<i>Uuxqt</i> running directory

**SEE ALSO**

*uucp*(1C), *uux*(1C), *L.cmds*(5), *USERFILE*(5), *uucico*(8C)



**NAME**

**vipw** – edit the password file

**SYNOPSIS**

**vipw**

**DESCRIPTION**

*Vipw* edits the password file while setting the appropriate locks, and does any necessary processing after the password file is unlocked. If the password file is already being edited, then you will be told to try again later. The *vi* editor will be used unless the environment variable **EDITOR** indicates an alternate editor. *Vipw* performs a number of consistency checks on the password entry for *root*, and will not allow a password file with a “mangled” root entry to be installed.

**SEE ALSO**

**passwd(1)**, **passwd(5)**, **adduser(8)**, **mkpasswd(8)**

**FILES**

**/etc/ptmp**

**NAME**

XNSrouted – NS Routing Information Protocol daemon

**SYNOPSIS**

```
/etc/XNSrouted [ -s ] [ -q ] [ -t ] [ logfile ]
```

**DESCRIPTION**

*XNSrouted* is invoked at boot time to manage the Xerox NS routing tables. The NS routing daemon uses the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries.

In normal operation *XNSrouted* listens for routing information packets. If the host is connected to multiple NS networks, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When *XNSrouted* is started, it uses the `SIOCGIFCONF` *ioctl* to find those directly connected interfaces configured into the system and marked “up” (the software loopback interface is ignored). If multiple interfaces are present, it is assumed the host will forward packets between networks. *XNSrouted* then transmits a *request* packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for *request* and *response* packets from other hosts.

When a *request* packet is received, *XNSrouted* formulates a reply based on the information maintained in its internal tables. The *response* packet generated contains a list of known routes, each marked with a “hop count” metric (a count of 16, or greater, is considered “infinite”). The metric associated with each route returned provides a metric *relative to the sender*.

*Response* packets received by *XNSrouted* are used to update the routing tables if one of the following conditions is satisfied:

- (1) No routing table entry exists for the destination network or host, and the metric indicates the destination is “reachable” (i.e. the hop count is not infinite).
- (2) The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.
- (3) The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.
- (4) The new route describes a shorter route to the destination than the one currently stored in the routing tables; the metric of the new route is compared against the one stored in the table to decide this.

When an update is applied, *XNSrouted* records the change in its internal tables and generates a *response* packet to all directly connected hosts and networks. *Routed* waits a short period of time (no more than 30 seconds) before modifying the kernel's routing tables to allow possible unstable situations to settle.

In addition to processing incoming packets, *XNSrouted* also periodically checks the routing table entries. If an entry has not been updated for 3 minutes, the entry's metric is set to infinity and marked for deletion. Deletions are delayed an additional 60 seconds to insure the invalidation is propagated to other routers.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks.

Supplying the `-s` option forces *XNSrouted* to supply routing information whether it is acting as an internetwork router or not. The `-q` option is the opposite of the `-s` option. If the `-t` option is specified, all packets sent or received are printed on the standard output. In addition, *XNSrouted* will not divorce itself from the controlling terminal so that interrupts from

the keyboard will kill the process. Any other argument supplied is interpreted as the name of file in which *XNSrouted*'s actions should be logged. This log contains information about any changes to the routing tables and a history of recent messages sent and received which are related to the changed route.

**SEE ALSO**

"Internet Transport Protocols", XSI 028112, Xerox System Integration Standard.  
idp(4P)





**Installing and Operating 4.3BSD on the VAX**  
**April 1, 1986**

*Michael J. Karels*

*James M. Bloom*

*Marshall Kirk McKusick*

*Samuel J. Leffler*

*William N. Joy*

Computer Systems Research Group  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California 94720  
(415) 642-7780

**ABSTRACT**

This document contains instructions for the installation and operation of the 4.3BSD release of the VAX\* UNIX\*\* system, as distributed by The University of California at Berkeley.

It discusses procedures for installing UNIX on a new VAX, and for upgrading an existing 4.2BSD VAX UNIX system to the new release. An explanation of how to lay out file systems on available disks, how to set up terminal lines and user accounts, and how to do system-specific tailoring is provided. A description of how to install and configure the networking facilities included with 4.3BSD is included. Finally, the document details system operation procedures: shutdown and startup, hardware error reporting and diagnosis, file system backup procedures, resource control, performance monitoring, and procedures for recompiling and reinstalling system software.

---

\* DEC, VAX, IDC, SBI, UNIBUS and MASSBUS are trademarks of Digital Equipment Corporation.

\*\* UNIX is a Trademark of Bell Laboratories.

April 16, 1986

## 1. INTRODUCTION

This document explains how to install the 4.3BSD release of the Berkeley version of UNIX for the VAX on your system. Because of the file system organization used in 4.3BSD, if you are not currently running 4.2BSD you will have to do a full bootstrap from the distribution tape. The procedure for performing a full bootstrap is outlined in chapter 2. The process includes booting standalone utilities from tape to format a disk if necessary, then to copy a small root filesystem image onto a swap area. This filesystem is then booted and used to extract a dump of a standard root filesystem. Finally, that root filesystem is booted, and the remainder of the system binaries and sources are read from the archives on the tape(s).

The technique for upgrading a 4.2BSD system is described in chapter 3 of this document. As 4.3BSD is upward-compatible with 4.2BSD, The upgrade procedure involves extracting a new set of system binaries onto new root and /usr filesystems. The sources are then extracted, and local configuration files are merged into the new system. 4.2BSD user filesystems may up upgraded in place, and 4.2BSD binaries may be used with 4.3BSD in the course of the conversion. It is desirable to recompile most local software after the conversion, as there are many changes and performance improvements in the standard libraries.

### 1.1. Hardware supported

This distribution can be booted on a VAX 8650, VAX 8600, VAX-11/785, VAX-11/780, VAX-11/750, VAX-11/730 or VAX-11/725 cpu with any of the following disks:

DEC MASSBUS:	RM03, RM05, RM80, RP06, RP07
EMULEX MASSBUS:	AMPEX Capricorn, 9300, CDC 9766, 9775, FUJITSU 2351 Eagle
DEC UNIBUS:	RK07, RL02, RA80, RA81, RA60, RC25
EMULEX SC-21V, SC-31	AMPEX DM980, Capricorn, 9300,
UNIBUS*:	CDC 9762, 9766, FUJITSU 160M, 330M
EMULEX SC-31 UNIBUS*:	FUJITSU 2351 Eagle
DEC IDC:	R80, RL02

The tape drives supported by this distribution are:

DEC MASSBUS:	TE16, TU45, TU77, TU78
EMULEX MASSBUS:	TC-7000
DEC UNIBUS:	TS11, TU80
EMULEX TC-11, AVIV UNIBUS:	KENNEDY 9300, STC, CIPHER
TU45 UNIBUS*:	SI 9700

The tapes and disks may be on any available UNIBUS or MASSBUS adapter at any slot with the proviso that the tape device must be slave number 0 on the formatter if it is a MASSBUS tape drive.

This distribution does not support the DEC CI780 or the HSC50 disk controller. As such, this distribution will not boot on the standard VAX 8600 and VAX 8650 cluster configurations. You will need to configure your system to use only UNIBUS and MASSBUS disk and tape devices.

\* Other UNIBUS controllers and drives may be easily usable with the system, but will likely require minor modifications to the system to allow bootstrapping. The EMULEX disk and SI tape controllers, and the drives shown here are known to work as bootstrap devices.

## 1.2. Distribution format

The basic distribution contains the following items:

- (3) 1600bpi 2400' magnetic tapes, or
- (1) 6250bpi 2400' magnetic tape, and
- (1) TU58 console cassette, and
- (1) RX01 console floppy disk.

Installation on any machine requires a tape unit. Since certain standard VAX packages do not include a tape drive, this means one must either borrow one from another VAX system or one must be purchased separately. The console media distributed with the system are not suitable for use as the standard console media; their intended use is only for installation.

The distribution does not fit on several standard VAX configurations that contain only small disks. If your hardware configuration does not provide at least 75 Megabytes of disk space you can still install the distribution, but you will probably have to operate without source for the user level commands and, possibly, the source for the operating system. The RK07-only distribution format once provided by our group is no longer available. Further, no attempt has ever been made to install the system on the standard VAX-11/730 hardware configuration from DEC that contains only dual RL02 disk drives (though the distribution tape may be bootstrapped on an RL211 controller and the system provides support for RL02 disk drives either on an IDC or an RL211). The labels on the distribution tape(s) show the amount of disk space each tape file occupies, these should be used in selecting file system layouts on systems with little disk space.

If you have the facilities, it is a good idea to copy the magnetic tape(s) in the distribution kit to guard against disaster. The tapes are 9-track 1600 BPI or 6250 BPI and contain some 512-byte records followed by many 10240-byte records. There are interspersed tape marks; end-of-tape is signaled by a double end-of-file.

The basic bootstrap material is present in three short files at the beginning of the first tape. The first file on the tape contains preliminary bootstrapping programs. This is followed by a binary image of a 2 megabyte "mini root" file system. Following the mini root file is a full dump of the root file system (see *dump(8)\**). Additional files on the tape(s) contain tape archive images (see *tar(1)*). See Appendix A for a description of the contents and format of the tape(s). One file contains software contributed by the user community; refer to the accompanying documentation for a description of its contents and an explanation of how it should be installed.

## 1.3. VAX hardware terminology

This section gives a short discussion of VAX hardware terminology to help you get your bearings.

If you have MASSBUS disks and tapes it is necessary to know the MASSBUS that they are attached to, at least for the purposes of bootstrapping and system description. The MASSBUSES can have up to 8 devices attached to them. A disk counts as a device. A tape *formatter* counts as a device, and several tape drives may be attached to a formatter. If you have a separate MASSBUS adapter for a disk and one for a tape then it is conventional to put the disk as unit 0 on the MASSBUS with the lowest "TR" number, and the tape formatter as unit 0 on the next MASSBUS. On a 11/780 this would correspond to having the disk on "mba0" at "tr8" and the tape on "mba1" at "tr9". Here the MASSBUS adapter with the lowest TR number has been called "mba0" and the one with the next lowest number is called "mba1".

To find out the MASSBUS that your tape and disk are on you can examine the cabling and the unit numbers or your site maintenance guide. Do not be fooled into thinking that the number on the front of the tape drive is a device number; it is a *slave* number, one of several possible tapes on the single tape formatter. For bootstrapping, the slave number must be 0. The formatter unit number

\* References of the form X(Y) mean the subsection named X in section Y of the UNIX programmer's manual.

may be anything distinct from the other numbers on the same MASSBUS, but you must know what it is.

The MASSBUS devices are known by several different names by DEC software and by UNIX. At various times it is necessary to know both names. There is, of course, the name of the device like "RM03" or "RM80"; these are easy to remember because they are printed on the front of the device. DEC also names devices based on the driver name in the system using a convention that reflects the interconnect topology of the machine. The first letter of such a name is a "D" for a disk, the second letter depends on the type of the drive, "DR" for RM03, RM05, and RM80's, "DB" for RP06's. The next letter is related to the interconnect; DEC calls the first MASSBUS or UNIBUS adapter "A", the second "B", etc. Thus, "DRA" is a RM drive on the first MASSBUS adapter. Finally, the name ends in a digit corresponding to the unit number for the device on the MASSBUS, i.e. "DRA0" is a disk at the first device slot on the first MASSBUS adapter and is an RM disk.

#### 1.4. UNIX device naming

UNIX has a set of names for devices which are different from the DEC names for the devices, viz.:

RM/RP disks	hp
TE/TU tapes	ht
TU78 tape	mt

The normal standalone system, used to bootstrap the full UNIX system, uses device names:

$xx(y,z)$

where  $xx$  is either `hp`, `ht`, or `mt`. The value  $y$  specifies the MASSBUS to use and also the device. It is computed as

$$8 * mba + device$$

Thus `mba0` device 0 would have a  $y$  value of 0 while `mba1` device 0 would have a  $y$  value of 8. The  $z$  value is interpreted differently for tapes and disks: for disks it is a disk *partition* (in the range 0-7), and for tapes it is a file number on the tape.

Each UNIX physical disk is divided into 8 logical disk partitions, each of which may occupy any consecutive cylinder range on the physical device. The cylinders occupied by the 8 partitions for each drive type are specified in section 4 of the programmers manual and in the disk description file `/etc/disktab` (c.f. `disktab(5)`).\* Each partition may be used for either a raw data area such as a paging area or to store a UNIX file system. It is conventional for the first partition on a disk to be used to store a root file system, from which UNIX may be bootstrapped. The second partition is traditionally used as a paging area, and the rest of the disk is divided into spaces for additional "mounted file systems" by use of one or more additional partitions.

The third logical partition of each physical disk also has a conventional usage: it allows access to the entire physical device, including the bad sector forwarding information recorded at the end of the disk (one track plus 126 sectors). It is occasionally used to store a single large file system or to access the entire pack when making a copy of it on another. Care must be taken when using this partition not to overwrite the last few tracks and thereby clobber the bad sector information.

The disk partitions have names in the standalone system of the form "`hp(y,z)`" with varying  $y$  as described above. Thus partition 1 of a RM05 on `mba0` at drive 0 would be "`hp(0,1)`". When not running standalone, this partition would normally be available as "`/dev/hp0b`". Here the prefix "`/dev`" is the name of the directory where all "special files" normally live, the "`hp`" serves an obvious

\* It is possible to change the partitions by changing the code for the table in the disk driver; it is often desirable to do this, therefore these tables should be read off each pack; they may be in a future version of the system.



purpose, the "0" identifies this as a partition of hp drive number "0" and the "b" identifies this as the second partition.

In all simple cases, a drive with unit number 0 (in its unit plug on the front of the drive) will be called unit 0 in its UNIX file name. This is not, however, strictly necessary, since the system has a level of indirection in this naming. If there are multiple controllers, the disk unit numbers will normally be counted sequentially across controllers. This can be taken advantage of to make the system less dependent on the interconnect topology, and to make reconfiguration after hardware failure extremely easy. We will not discuss that now.

Returning to the discussion of the standalone system, we recall that tapes also took two integer parameters. In the normal case where the tape formatter is unit 0 on the second mba (mba1), the files on the tape have names "ht(8,0)", "ht(8,1)", etc. Here "file" means a tape file containing a single data stream. The distribution tape(s) have data structures in the tape files and though the tape(s) contain only 9 tape files, they contain several thousand UNIX files.

For the UNIBUS, there are also conventional names. The important DEC names to know are DM?? for RK07 drives and DU?? for drives on a UDA50. For example, RK07 drive 0 on a controller on the first UNIBUS on the machine is "DMA0". UNIX calls such a device an "hk" and the standalone name for the first partition of such a device is "hk(0,0)". The first number is calculated from the drive number and UNIBUS adapter as

$$8 * uba + drive$$

If the controller were on the second UNIBUS its name would be "hk(8,0)". If we wished to access the first partition of an RK07 drive 1 on uba0 we would use "hk(1,0)".

The UNIBUS disk and tape names used by UNIX are:

RK disks	hk
TS tapes	ts
UDA disks	ra
RC25 disks	ra
IDC disks	rb
SMD disks	up
TM tapes	tm
TMSCP tapes	tmscp
TU tapes	ut

Here SMD disks are disks on an RM-emulating controller on the UNIBUS, and TM tapes are tapes on a controller that emulates the DEC TM11. TU tapes are tapes on a UNIBUS controller that emulates the DEC TU45. IDC disks are disks on an 11/730 Integral Disk Controller. TS tapes are tapes on a controller compatible with the DEC TS11 (e.g. a TU80). The naming conventions for partitions in UNIBUS disks and files in UNIBUS tapes are the same as those for MASSBUS disks and tapes.

### 1.5. UNIX devices: block and raw

UNIX makes a distinction between "block" and "raw" (character) devices. Each disk has a block device interface where the system makes the device byte addressable and you can write a single byte in the middle of the disk. The system will read out the data from the disk sector, insert the byte you gave it and put the modified data back. The disks with the names "/dev/xx0a", etc are block devices. There are also raw devices available. These have names like "/dev/rxx0a", the "r" here standing for "raw". Raw devices bypass the buffer cache and use DMA directly to/from the program's I/O buffers; they are normally restricted to full-sector transfers. In the bootstrap procedures we will often suggest using the raw devices, because these tend to work faster. Raw devices are used when making new filesystems, when checking unmounted filesystems, or for copying quiescent filesystems. The block devices are used to mount file systems, or when operating on a mounted filesystem such as the root.

You should be aware that it is sometimes important whether to use the character device (for efficiency) or not (because it wouldn't work, e.g. to write a single byte in the middle of a sector). Don't change the instructions by using the wrong type of device indiscriminately.

April 16, 1986

## 2. BOOTSTRAP PROCEDURE

This section explains the bootstrap procedure that can be used to get the kernel supplied with this distribution running on your machine. If you are not currently running 4.2BSD you will have to do a full bootstrap. Chapter 3 describes how to upgrade an existing 4.2BSD system. An understanding of the operations used in a full bootstrap is very helpful in performing an upgrade as well. In either case, it is highly desirable to read and understand the remainder of this document before proceeding.

### 2.1. Converting pre-4.2BSD Systems

The file system format was changed between 3BSD and 4.0BSD, and again between 4.1BSD and 4.2BSD. At a minimum you will have to dump your old file systems, and then restore them onto the 4.3BSD file system. Sites running 3BSD or 32/V may be able to modify the *restore* program to understand the old 512 byte block file system, but this has never been tried. The dump format used in 4.0BSD and 4.1BSD is backward-compatible with that used in 4.3BSD (which is unchanged from 4.2BSD). That is, the 4.3BSD *restore* program understands how to read 4.0BSD and 4.1BSD dump tapes, although 4.3BSD dump tapes cannot be restored under 4.0BSD or 4.1BSD. It is also desirable to make a convenient copy of system configuration files for use as guides when setting up the new system; the list of files to save from 4.2BSD systems in chapter 3 may be used as a guideline.

The first step is to dump your file systems with *dump*(8). For the utmost of safety this should be done to magtape. However, if you enjoy gambling with your life (or you have a VERY friendly user community) and you have enough disk space, you can try converting your file systems while copying to a new disk partition by piping the output of *dump* directly into *restore* after bringing up 4.3BSD. If you select the latter tack, a version of the 4.1BSD dump program that runs under 4.3BSD is provided in */etc/dump.4.1*. Beware that file systems created under 4.3BSD can use about 5-10% more disk space for file system related information than under 4.1BSD. Thus, before dumping each file system it is a good idea to remove any files that may be easily regenerated. Since most all programs will likely be recompiled under the new system your best bet is to remove any object files. File systems with at least 10% free space on them should restore into an equivalently sized 4.3BSD file system without problem.

### 2.2. Booting from tape

The tape bootstrap procedure used to create a working system involves the following major steps:

- 1) Format a disk pack with the *format* program.
- 2) Copy a "mini root" file system from the tape onto the swap area of the disk.
- 3) Boot the UNIX system on the "mini root".
- 4) Restore the full root file system using *restore*(8).
- 5) Build a console floppy, cassette, or RL02 pack for bootstrapping.
- 6) Reboot the completed root file system.
- 7) Build and restore the /usr file system from tape with *tar*(1).
- 8) Extract the system and utility files and contributed software as desired.

Certain of these steps are dependent on your hardware configuration. Formatting the disk pack used for the root file system may require using the DEC standard formatting programs. Also, if you are bootstrapping the system on an 11/750, no console cassette is created.

Bootstrapping an 8650 or 8600 is a bit more difficult than bootstrapping the other machines. The procedures for loading the toggle program and reading the tape bootstrap monitor described in Appendix B must be used if you do not have access to a console RL02 pack with a UNIX bootstrap. Such a pack may be made on an 8600 already running UNIX, or on another 4.3BSD system with an

RL02 drive using the procedures in 4.1.1. One may be required to enter the toggle program more than once. After the bootstrap monitor is loaded, device addresses will be the same as if the machine were an 11/780 or 11/785.

The following sections describe the above steps in detail. In these sections references to disk drives are of the form *xx(n,m)* and references to files on tape drives are of the form *yy(n,m)* where *xx* and *yy* are names described in section 1.4 and *n* and *m* are the unit and offset numbers described in section 1.4. Commands you are expected to type are shown in Roman, while that information printed by the system is shown emboldened. Throughout the installation steps the reboot switch on an 11/785, 11/780 or 11/730 should be set to off; on an 8650, 8600 or 11/750 set the power-on action to halt. (In normal operation an 11/785, 11/780 or 11/730 will have the reboot switch on and an 8650, 8600 or 11/750 will have the power-on action set to reboot/restart.)

If you encounter problems while following the instructions in this part of the document, refer to Appendix C for help in troubleshooting.

### 2.2.1. Step 1: formatting the disk

All disks used with 4.3BSD should be formatted to insure the proper handling of physically corrupted disk sectors. If you have DEC disk drives, you should use the standard DEC formatter to format your disks. If not, the *format* program included in the distribution, or a vendor supplied formatting program, may be used to format disks. The *format* program is capable of formatting any of the following supported distribution devices:

EMULEX MASSBUS:	AMPEX Capricorn, 9300, CDC 9766, 9775,
	FUJITSU 330M, 2351 Eagle
EMULEX SC-21V, SC-31	AMPEX 9300, Capricorn, CDC 9730, 9766,
UNIBUS:	FUJITSU 160M, 330M
EMULEX SC-31 UNIBUS:	FUJITSU 2351 Eagle

If you have run a pre-4.1BSD version of UNIX on the packs you are planning to use for bootstrapping it is likely that the bad sector information on the packs has been destroyed, since it was accessible as normal data in the last several tracks of the disk. You should therefore run the formatter again to make sure the information is valid.

On an 11/750, to use a disk pack as a bootstrap device, sectors 0 through 15, the disk sectors in the file "/boot" (the program that loads the system image), and the file system indices that lead to this file must not have any errors. On an 8650, 8600, 11/785, 11/780 or 11/730, the "boot" program is loaded from the console medium and includes device drivers for the "hp" and "up" disks that do ECC correction and bad sector forwarding; consequently, on these machines the system may be bootstrapped on these disks even if the disk is not error free in critical locations. In general, if the first 15884 sectors of your disk are clean you are safe; if not you can take your chances.

To load the *format* program, insert the distribution TU58 cassette or RX01 floppy disk in the appropriate console device (on the 11/730 use cassette 0) and do the following steps.

If you have an 8650 or 8600 start the bootstrap monitor using the procedure described in Appendix B. Then give the command:

```
= format
```

If you have an 11/785 or 11/780 give the commands:

```
>>> HALT
>>> UNJAM
>>> INIT
>>> LOAD FORMAT
>>> START 2
```

If you have an 11/750 give the commands:

```
>>> I
>>> B DDA0
= format
```

If you have an 11/730 give the commands:

```
>>> H
>>> I
>>> L DD0:FORMAT
>>> S 2
```

The *format* program should now be running and awaiting your input:

Disk format/check utility

Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?

If you made a mistake loading the program off the TU58 cassette or using the bootstrap monitor loaded for the 8650 or 8600 the “=” prompt should reappear and you can retype the program name. If something else happened, you may have a bad distribution cassette or floppy, or your hardware may be broken; refer to Appendix C for help in troubleshooting. If you are unable to load programs off the distributed medium, consult Appendix B for an alternate (more painful) approach.

*Format* will create sector headers and verify the integrity of each sector formatted by using the disk controller's “write check” command. Remember *format* runs only on the up and hp drives listed above. *Format* will prompt for the information required as shown below. Questions with default answers appear with the default in parentheses at the prompt; a carriage return will take the default. If you err in answering questions, “Delete” erases the last character typed, and “U” erases the current input line.

April 16, 1986

```

Enable debugging (0=none, 1=bse, 2=ecc, 3=bse+ecc)?
Device to format? xx(0,0)
...(the old bad sector table is read; ignore any errors that occur here)...
Formatting drive xx0 on adaptor 0: verify (yes/no)? yes
Device data: #cylinders=842, #tracks=20, #sectors=48
Starting cylinder (0):
Starting track (0):
Ending cylinder (841):
Ending track (19):
Available test patterns are:
    1 - (f00f) RH750 worst case
    2 - (ec6d) media worst case
    3 - (a5a5) alternating 1's and 0's
    4 - (ffff) Severe burnin (up to 48 passes)
Pattern (one of the above, other to restart)? 2
Maximum number of bit errors to allow for soft ECC (3):
Start formatting...make sure the drive is online
...(soft ecc's and other errors are reported as they occur)...
...(if 4 write check errors were found, the program terminates like this)...
Errors:
Bad sector: 0
Write check: 4
Hard ECC: 0
Other hard: 0
Marked bad: 0
Skipped: 0
Total of 4 hard errors revectorred.
Writing bad sector table at block 524256
...(524256 is the block # of the first block in the bad sector table)...
Done

```

Once the root device has been formatted, *format* will prompt for another disk to format. Halt the machine by typing "control-P" and "H" (the "H" is necessary only on an 11/785 or 11/780, but does not hurt on the other machines).

```

Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?^P
>>> H

```

It may be necessary to format other drives before constructing file systems on them; this can be done at a later time with the steps just performed. *Format* can also be used in an extended test mode (pattern 4) that uses numerous test patterns in up to 48 passes to detect as many disk surface errors as possible; this test may be run for many hours, depending on the CPU and controller. On an 11/780, this can be sped up significantly by setting the clock fast. It may be run for some number of passes, then either terminated or continued according to the errors found to that point.

#### 2.2.2. Step 2: copying the mini-root file system

The second step is to run a simple program, *copy*, which copies a small root file system into the second partition of the disk. This file system will serve as the base for creating the actual root file system to be restored. The version of the operating system maintained on the "mini-root" file system understands that it should not swap on top of itself, thereby allowing double use of the disk partition. *Copy* is loaded just as the *format* program was loaded; for example, on an 8650 or 8600, one needs to enter the toggle and the bootstrap monitor as described in Appendix B and then:

April 16, 1986

```

(copy mini root file system)
= copy
From: yy(y,1)                (unit y, second tape file)
To: xx(x,1)                  (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:

```

while for an 11/785 or 11/780:

```

(copy mini root file system)
>>> LOAD COPY
>>> START 2
From: yy(y,1)                (unit y, second tape file)
To: xx(x,1)                  (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:

```

or for an 11/750:

```

(copy mini root file system)
>>> B DDA0
= copy
From: yy(y,1)                (unit y, second tape file)
To: xx(x,1)                  (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:

```

and for an 11/730:

```

(copy mini root file system)
>>> L DD0:COPY
>>> S 2
From: yy(y,1)                (unit y, second tape file)
To: xx(x,1)                  (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:

```

(As above, 'delete' erases characters and "U" erases lines.)

### 2.2.3. Step 3: booting from the mini-root file system

You now have the minimal set of tools necessary to create a root file system and restore the file system contents from tape. To access this file system load the bootstrap program and boot the version of unix that has been placed in the "mini-root":

```

(follow the procedure in Appendix B to load the bootstrap monitor)

(load bootstrap program)
= boot
Boot
: xx(x,1)vmunix              (bring in vmunix off mini root)

```

or, on an 11/780 or 11/785:

April 16, 1986

```
(load bootstrap program)
>>> BOOT ANY
Boot
: xx(x,1)vmunix          (bring in vmunix off mini root)
```

or, on an 11/750:

```
(load bootstrap program)
>>> B DDA0
= boot
Boot
: xx(x,1)vmunix          (bring in vmunix off mini root)
```

or, on an 11/730:

```
(load bootstrap program)
>>> L DD0:BOOT
>>> D RB 3
>>> S 2
Boot
: xx(x,1)vmunix          (bring in vmunix off mini root)
(As above, 'delete' erases characters and 'U' erases lines.)
```

The standalone boot program should then read the system from the mini root file system you just created, and the system should boot:

```
271944+78848+92812 start 0x12e8
4.3 BSD UNIX #1: Wed Apr 9 23:33:59 PST 1985
karels@monet.berkeley.edu:/usr/src/sys/GENERIC
real mem = xxx
avail mem = yyy
... information about available devices ...
root device?
```

The first three numbers are printed out by the bootstrap programs and are the sizes of different parts of the system (text, initialized and uninitialized data). The system also allocates several system data structures after it starts running. The sizes of these structures are based on the amount of available memory and the maximum count of active users expected, as declared in a system configuration description. This will be discussed later.

UNIX itself then runs for the first time and begins by printing out a banner identifying the release and version of the system that is in use and the date that it was compiled.

Next the *mem* messages give the amount of real (physical) memory and the memory available to user programs in bytes. For example, if your machine has only 512K bytes of memory, then xxx will be 520192, 4096 bytes less than 512K. The system reserves the last 4096 bytes of memory for use in error logging and doesn't count it as part of real memory.

The messages that come out next show what devices were found on the current processor. These messages are described in *autoconf(4)*. The distributed system may not have found all the communications devices you have (dh's, dz's, etc.), or all the mass storage peripherals you have especially if you have more than two of anything. You will correct this soon, when you create a description of your machine from which to configure UNIX. The messages printed at boot here contain much of the information that will be used in creating the configuration. In a correctly configured system most of the information present in the configuration description is printed out at boot time as the system verifies that each device is present.

April 16, 1986



The “root device?” prompt was printed by the system and is now asking you for the name of the root file system to use. This happens because the distribution system is a *generic* system. It can be bootstrapped on any VAX cpu and with its root device and paging area on any available disk drive. You should respond to the root device question with *xx0\**. This response supplies two pieces of information: first, *xx0* shows that the disk it is running on is drive 0 of type *xx*, secondly the “\*” shows that the system is running “atop” the paging area. The latter is most important, otherwise the system will attempt to page on top of itself and chaos will ensue. You will later build a system tailored to your configuration that will not ask this question when it is bootstrapped.

```
root device? xx0*
```

```
WARNING: preposterous time in file system — CHECK AND RESET THE DATE!
```

```
erase ^?, kill ^U, intr ^C
```

```
#
```

The “erase ...” message is part of */.profile* that was executed by the root shell when it started. This message is present to remind you that the line character erase, line erase, and interrupt characters are set to be what is standard on DEC systems; this insures that things are consistent with the DEC console interface characters.

#### 2.2.4. Step 4: restoring the root file system

UNIX is now running, and the ‘UNIX Programmer’s manual’ applies. The ‘#’ is the prompt from the shell, and lets you know that you are the super-user, whose login name is “root”. To complete installation of the bootstrap system two steps remain. First, the root file system must be created, and second a boot floppy or cassette must be constructed.

To create the root file system the shell script “xtr” should be run as follows:

```
# disk=xx0 type=tt tape=yy xtr
```

where *xx0* is the name of the disk on which the root file system is to be restored (unit 0), *tt* is the type of drive on which the root file system is to be restored (see the table below), and *yy* is the name of the tape drive on which the distribution tape is mounted.

If the root file system is to reside on a disk other than unit 0 (as the information printed out during autoconfiguration shows), you will have to create the necessary special files in */dev* and use the appropriate value. For example, if the root should be placed on *hp1*, you must create */dev/rhp1a* and */dev/hp1a* using *mknod(8)*.

Drive	Type	Drive	Type
DEC RM03	type=rm03	DEC RM05	type=rm05
DEC RM80	type=rm80	DEC RP06	type=rp06
DEC RP07	type=rp07	DEC RK07	type=rk07
DEC RA80	type=ra80	DEC RA60	type=ra60
DEC RA81	type=ra81	DEC R80	type=rb80
CDC 9766	type=9766	CDC 9775	type=9775
AMPEX 300M	type=9300	AMPEX 330M	type=capricorn
FUJITSU 160M	type=fuji160	FUJITSU 330M	type=capricorn
FUJITSU 404M	type=eagle		

This will generate many messages regarding the construction of the file system and the restoration of the tape contents, but should eventually stop with the messages:

...  
Root filesystem extracted

If this is an 8650 or 8600, update the console RL02  
If this is a 780 or 785, update the floppy  
If this is a 730, update the cassette  
#

#### 2.2.5. Step 5: creating a boot floppy or cassette

If the machine is an 8650, 8600, 11/785, 11/780 or 11/730, a boot floppy, cassette, or console RL02 should be constructed according to the instructions in chapter 4. For 11/750's, bootstrapping is performed by using a boot prom and special code located in sectors 0-15 of the root file system. The *newfs* program automatically installs the needed code, so you may continue with the next step. On an 11/785 or 11/780 with interleaved memory, or other configurations that require alteration of the standard boot files, this step may be left for later.

#### 2.2.6. Step 6: rebooting the completed root file system

With the above work completed, all that is left is to reboot:

```
#sync                                (synchronize file system state)
#^P                                (halt machine)
>>> HALT                            (for 11/785's or 11/780's only)
>>> UNJAM                          (for 8650's, 8600's, 11/785's or 11/780's only)
>>> I                              (initialize processor state)
>>> B xxS                          (on an 11/750, use B/2)
...(boot program is eventually loaded)...
Boot
: xx(x,0)vmunix                    (vmunix brought in off root)
271944+78848+92812 start 0x12e8
4.3 BSD UNIX #1: Wed Apr 9 23:33:59 PST 1985
karels@monet.berkeley.edu:/usr/src/sys/GENERIC
real mem = xxx
avail mem = yyy
... information about available devices ...
root on xx0
WARNING: preposterous time in file system --- CHECK AND RESET THE DATE!
erase ^?, kill ^U, Intr ^C
#
```

If the root device selected by the kernel is not correct, it is necessary to reboot again using the option to ask for the root device. On the 11/750, use B/3; on the other processors, use BOOT ANY. At the prompt from the bootstrap, use the same device specification above: *xx(x,0)vmunix*. Then, to the question "root device?," respond with *xx0*. See section 6.1 and appendix C if the system does not reboot properly.

The system is now running single user on the installed root file system. The next section tells how to complete the installation of distributed software on the */usr* file system.

#### 2.2.7. Step 7: setting up the */usr* file system

First set a shell variable to the name of your disk, so the commands we give will work regardless of the disk you have; do one of the following:

April 16, 1986

```
# disk=hp      (if you have an RP06, RM03, RM05, RM80, or other MASSBUS drive)
# disk=hk      (if you have RK07s)
# disk=ra      (if you have UDA50 storage module drives)
# disk=up      (if you have UNIBUS storage module drives)
# disk=rb      (if you have IDC storage module drives)
```

The next thing to do is to extract the rest of the data from the tape. You might wish to review the disk configuration information in section 4.4 before continuing; the partitions used below are those most appropriate in size. Find the disk you have in the following table and execute the commands in the right hand portion of the table:

DEC RM03	# name=hp0g; type=rm03
DEC RM05	# name=hp0g; type=rm05
DEC RM80	# name=hp0g; type=rm80
DEC RP06	# name=hp0g; type=rp06
DEC RP07	# name=hp0h; type=rp07
DEC RK07	# name=hk0g; type=rk07
DEC RA80	# name=ra0h; type=ra80
DEC RA60	# name=ra0h; type=ra60
DEC RA81	# name=ra0h; type=ra81
DEC R80	# name=rb0h; type=rb80
UNIBUS CDC 9766	# name=up0g; type=9766
UNIBUS AMPEX 300M	# name=up0g; type=9300
UNIBUS AMPEX 330M	# name=up0g; type=capricorn
UNIBUS FUJITSU 160M	# name=up0g; type=fuji160
UNIBUS FUJITSU 330M	# name=up0g; type=capricorn
UNIBUS FUJITSU 404M	# name=up0h; type=eagle
MASSBUS CDC 9766	# name=hp0g; type=9766
MASSBUS AMPEX 300M	# name=hp0g; type=9300
MASSBUS AMPEX 330M	# name=hp0g; type=capricorn
MASSBUS FUJITSU 330M	# name=hp0g; type=capricorn
MASSBUS FUJITSU 404M	# name=hp0h; type=eagle

Find the tape you have in the following table and execute the commands in the right hand portion of the table:

DEC TE16/TU45/TU77	# cd /dev; MAKEDEV ht0; sync
DEC TU78	# cd /dev; MAKEDEV mt0; sync
DEC TS11	# cd /dev; MAKEDEV ts0; sync
EMULEX TC11	# cd /dev; MAKEDEV tm0; sync
SI 9700	# cd /dev; MAKEDEV ut0; sync

Then execute the following commands:

April 16, 1986

```
# date yymmddhhmm          (set date, see date(1))
....
# passwd root                (set password for super-user)
New password:                (password will not echo)
Retype new password:
# hostname mysitename        (set your hostname)
# newfs $(name) $(type)      (create empty user file system)
                             (this takes a few minutes)
# mount /dev/$(name) /usr     (mount the usr file system)
# cd /usr                    (make /usr the current directory)
# mt fsf
# tar xpbf 20 /dev/rmt12      (extract all of usr except usr/src)
                             (this takes about 15-20 minutes)
```

If the tape had been rewound or positioned incorrectly before the *tar*, it may be repositioned by the following commands.

```
# mt rew
# mt fsf 3
```

The data on the fourth tape file has now been extracted. If you are using 1600bpi tapes, the first reel of the distribution is no longer needed; the remainder of the installation procedure uses the second reel of tape that should be mounted in place of the first. The first instruction below is ignored if using 1600bpi tapes. The installation procedure continues from this point on the 6250bpi tape.

```
# mt fsf
# mkdir src                  (make directory for source)
# mkdir src/sys              (make directory for system source)
# cd src/sys                 (make /usr/sys the current directory)
# tar xpbf 20 /dev/rmt12     (extract the system source)
                             (this takes about 5-10 minutes)
# cd /                       (back to root)
# chmod 755 /usr /usr/src /usr/src/sys
# rm -f sys                  (make a symbolic link to the system source)
# ln -s usr/src/sys sys      (unmount /usr)
# umount /dev/$(name)
```

You can check the consistency of the /usr file system by doing

```
# fsck /dev/r$(name)
```

The output from *fsck* should look something like:

```
** /dev/rxx0h
** Last Mounted on /usr
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
671 files, 3497 used, 137067 free (75 frags, 34248 blocks)
```

If there are inconsistencies in the file system, you may be prompted to apply corrective action; see the document describing *fsck* for information.

To use the /usr file system, you should now remount it by saying

April 16, 1986

```
# /etc/mount /dev/${name} /usr
```

You can then extract the source code for the commands (except on RK07's and RM03's this will fit in the /usr file system):

```
# cd /usr/src
# mt fsf
# tar xpb 20
```

If you get an error at this point, most likely it was a problem with tape positioning. You can reposition the tape by rewinding it and then skipping over the files already read (see *mt(1)*).

#### 2.2.8. Additional software

There are three extra tape files on the distribution tape(s) which have not been installed to this point. They are a font library for use with Varian and Versatec printers, the Ingres database system, and user contributed software. All three tape files are in *tar(1)* format and can be installed by positioning the tape using *mt(1)* and reading in the files as was done for /usr/src above. As distributed, the fonts should be placed in a directory /usr/lib/vfont, the Ingres system should be placed in /usr/ingres, and the user contributed software should be placed in /usr/src/new. The exact contents of the user contributed software is given in a separate document.

#### 2.3. Additional conversion information

After setting up the new 4.3BSD filesystems, you may restore the user files that were saved on tape before beginning the conversion. Note that the 4.3BSD *restore* program does its work on a mounted file system using normal system operations (unlike the older *restor* that accessed the raw file system device and deposited inodes in the appropriate locations on disk). This means that file system dumps may be restored even if the characteristics of the file system changed. To restore a dump tape for, say, the /a file system something like the following would be used:

```
# mkdir /a
# newfs hplg eagle
# mount /dev/hplg /a
# cd /a
# restore r
```

If you chose to convert filesystems while copying to a new disk area, do so by piping the output of *dump.4.1* directly into *restore* after bringing up 4.3BSD.

If *tar* images were written instead of doing a dump, you should be sure to use the 'p' option when reading the files back. No matter how you restore a file system, be sure and check its integrity with *fsck* when the job is complete.

To convert a compiler from 4.1BSD to 4.3BSD you should simply have to recompile and relink the various parts. If the processor is written in itself, for instance a PASCAL compiler written in PASCAL, the important step in converting is to save a working copy of the 4.1BSD binary before converting to 4.3BSD. Then, once the system has been changed over, the 4.1BSD binary should be used in the rebuilding process. To do this, you should enable the 4.1 compatibility option when you configure the kernel (see section 4.3).

If no working 4.1BSD binary exists, or the language processor uses some nonstandard system call, you will likely have to compile the language processor into an intermediate form, such as assembly language, on a 4.1BSD system, then bring the intermediate form to 4.3BSD for assembly and loading.

### 3. UPGRADING A 4.2BSD SYSTEM

Begin by reading the "Bugs Fixes and Changes in 4.3BSD" document to see what has changed since the last time you bootstrapped the system. If you have local system modifications to the kernel to install, look at the document "Changes to the Kernel in 4.3BSD" to get an idea of how the system changes will affect your local modifications.

If you are running 4.2BSD, upgrading your system involves replacing your kernel and system utilities. Binaries compiled under 4.2BSD will work without recompilation under 4.3BSD, though they may run faster if they are relinked. The easiest way to convert to 4.3BSD (depending on your file system configuration) is to create new root and /usr file systems from the distribution tape on unused disk partitions, boot the new system, and then copy any local utilities from your old root and /usr file systems into the new ones. All user file systems and binaries can be retained unmodified, except that the new *fsck* should be run before they are mounted (see below). 4.1BSD binary images can also run unchanged under 4.3BSD but only when the system is configured to include the "4.1BSD compatibility mode."\*

Section 3.1 lists the files to be saved as part of the conversion process. Section 3.2 describes the bootstrap process. Section 3.3 discusses the merger of the saved files back into the new system. Section 3.4 provides general hints on possible problems to be aware of when converting from 4.2BSD to 4.3BSD.

#### 3.1. Files to save

The easiest upgrade path from a 4.2BSD is to build new root and *usr* file systems on unused partitions, then copy or merge site specific files into their corresponding files on the new system. The following list enumerates the standard set of files you will want to save and suggests directories in which site specific files should be present. This list will likely be augmented with non-standard files you have added to your system. If you do not have enough space to create parallel file systems, you should create a *tar* image of the following files before the new file systems are created. In addition, you should do a full dump before rebuilding the file system to guard against missing something the first time around.

---

\* With "4.1BSD compatibility mode" system calls from 4.1BSD are either emulated or safely ignored. There are only two exceptions; programs that read directories or use the old jobs library will not operate properly. However, while 4.1BSD binaries will execute under 4.3BSD it is **STRONGLY RECOMMENDED** that the programs be recompiled under the new system.

/cshrc	†	root csh startup script
/login	†	root csh login script
/profile	†	root sh startup script
/rhosts	†	for trusted machines and users
/dev/MAKEDEV	‡	in case you added anything here
/dev/MAKEDEV.local	*	for making local devices
/etc/disktab	‡	in case you changed disk partition sizes
/etc/fstab	†	disk configuration data
/etc/ftpusers	†	for local additions
/etc/gateways	†	routing daemon database
/etc/gettytab	‡	getty database
/etc/group	*	group data base
/etc/hosts	†	for local host information
/etc/hosts.equiv	†	for local host equivalence information
/etc/networks	†	for local network information
/etc/passwd	*	user data base
/etc/printcap	†	line printer database
/etc/protocols	‡	in case you added any local protocols
/etc/rc	*	for any local additions
/etc/rc.local	*	site specific system startup commands
/etc/remote	†	auto-dialer configuration
/etc/services	‡	for local additions
/etc/syslog.conf	*	system logger configuration
/etc/securetty	*	for restricted list of ttys where root can log in
/etc/ttys	*	terminal line configuration data
/etc/ttytype	*	terminal line to terminal type mapping data
/etc/termcap	‡	for any local entries that may have been added
/lib	‡	for any locally developed language processors
/usr/dict/*	‡	for local additions to words and papers
/usr/hosts/MAKEHOSTS	†	for local changes
/usr/include/*	‡	for local additions
/usr/lib/aliases	†	mail forwarding data base
/usr/lib/crontab	*	cron daemon data base
/usr/lib/font/*	‡	for locally developed font libraries
/usr/lib/lib*.a	†	for locally libraries
/usr/lib/lint/*	‡	for locally developed lint libraries
/usr/lib/sendmail.cf	*	sendmail configuration
/usr/lib/tabset/*	‡	for locally developed tab setting files
/usr/lib/term/*	‡	for locally developed nroff drive tables
/usr/lib/tmac/*	‡	for locally developed troff/nroff macros
/usr/lib/uucp/*	†	for local uucp configuration files
/usr/man/man1	†	for manual pages for locally developed programs
/usr/msg	†	for current msgs
/usr/spool/*	†	for current mail, news, uucp files, etc.
/usr/src/local	†	for source for locally developed programs
/sys/conf/HOST	†	configuration file for your machine
/sys/conf/files.HOST	†	list of special files in your kernel
/*quotas	†	file system quota files

† Files that can be used from 4.2BSD without change.

‡ Files that need local modifications merged into 4.3BSD files.

\* Files that require special work to merge and are discussed below.

April 16, 1986

### 3.1.1. Installing 4.3BSD

The next step is to build a working 4.3BSD system. This can be done by following the steps in section 2 of this document for extracting the root and /usr file systems from the distribution tape onto unused disk partitions. If you have a running 4.2BSD system, you can also do this by using *dd*(1) to copy the "mini root" filesystem onto one disk partition, then use it to load the 4.3BSD root filesystem as in chapter 2. The root filesystem dump on the tape could also be extracted directly, although this will require an additional file system check after booting 4.3BSD to convert the new root filesystem. The exact procedure chosen will depend on the disk configuration and the number of suitable disk partitions that may be used. If there is insufficient space to load the new root and /usr filesystems before reusing the existing 4.2BSD partitions, it is strongly advised that you make full dumps of each filesystem on magtape before beginning. It is also desirable to run file system checks of all filesystems to be converted to 4.3BSD before shutting down 4.2BSD. If you are running an older system, you will have to dump and restore your file systems; see section 2.1 for some hints. In either case, this is an excellent time to review your disk configuration for possible tuning of the layout. Section 4.3 is required reading.

To ease the transition to new kernels, the 4.3BSD bootstrap routines now pass the identity of the boot device through to the kernel. The kernel then uses that device as its root file system. Thus, for example, if you boot from */dev/hp1a*, the kernel will use *hp1a* as its root file system. If */dev/hp1b* is configured as a swap partition, it will be used as the initial swap area, otherwise the normal primary swap area (*/dev/hp0b*) will be used. The 4.3BSD bootstrap is backward compatible with 4.2BSD, so you can replace your 4.2BSD bootstrap if you use it to boot your first 4.3BSD kernel.

Once you have extracted the 4.3BSD system and booted from it, you will have to build a kernel customized for your configuration. If you have any local device drivers, they will have to be incorporated into the new kernel. See section 4.2.3 and "Building 4.3BSD UNIX Systems with Config."

The disk partitions in 4.3BSD are the same as those in 4.2BSD, except for those on the DEC UDA50; see section 4.3.2 for details. If you have changed the disk partition sizes, be sure to make the necessary table changes and boot your custom kernel BEFORE trying to access any of your 4.2BSD file systems! After doing this if necessary, the remaining 4.2BSD filesystems may be converted in place. This is done by using the 4.3BSD version of *fsck*(8) on each filesystem and allowing it to make the necessary corrections. The new version of *fsck* is more strict about the size of directories than the version supplied with 4.2BSD. Thus the first time that it is run on a 4.2BSD file system, it will produce messages of the form:

DIRECTORY ...: LENGTH xx NOT MULTIPLE OF 512 (ADJUSTED)

Length "xx" will be the size of the directory; it will be expanded to the next multiple of 512 bytes. Note that file systems are otherwise completely compatible between 4.2BSD and 4.3BSD, though running a 4.3BSD file system under 4.2BSD may cause more of the above messages to be generated the next time it is *fsck*'ed on 4.3BSD.

### 3.2. Merging your files from 4.2BSD into 4.3BSD

When your system is booting reliably and you have the 4.3BSD root and /usr file systems fully installed you will be ready to continue with the next step in the conversion process, merging your old files into the new system.

If you saved the files on a *tar* tape, extract them into a scratch directory, say /usr/convert:

```
# mkdir /usr/convert
# cd /usr/convert
# tar x
```

The data files marked in the previous table with a dagger (†) may be used without change from the previous system. Those data files marked with a double dagger (§) have syntax changes or substantial enhancements. You should start with the 4.3BSD version and carefully integrate any local changes into the new file. Usually these local modifications can be incorporated without conflict into



the new file; some exceptions are noted below. The files marked with an asterisk (\*) require particular attention and are discussed below.

If you have any homegrown device drivers in `/dev/MAKEDEV.local` that use major device numbers reserved by the system you will have to modify the commands used to create the devices or alter the system device configuration tables in `/sys/vax/conf.c`. Otherwise `/dev/MAKEDEV.local` can be used without change from 4.2BSD.

System security changes require adding several new “well-known” groups to `/etc/group`. The groups that are needed by the system as distributed are:

name	number
wheel	0
daemon	1
kmem	2
sys	3
tty	4
operator	5
staff	10

Only users in the “wheel” group are permitted to `su` to “root”. Most programs that manage directories in `/usr/spool` now run `set-group-id` to “daemon” so that users cannot directly access the files in the spool directories. The special files that access kernel memory, `/dev/kmem` and `/dev/mem`, are made readable only by group “kmem”. Standard system programs that require this access are made `set-group-id` to that group. The group “sys” is intended to control access to system sources, and other sources belong to group “staff.” Rather than make user’s terminals writable by all users, they are now placed in group “tty” and made only group writable. Programs that should legitimately have access to write on user’s terminals such as *talk* and *write* now run `set-group-id` to “tty”. The “operator” group controls access to disks. By default, disks are readable by group “operator”, so that programs such as *df* can access the file system information without being `set-user-id` to “root”.

Several new users have also been added to the group of “well-known” users in `/etc/passwd`. The current list is:

name	number
root	0
daemon	1
operator	2
uucp	66
nobody	32767

The “daemon” user is used for daemon processes that do not need root privileges. The “operator” user-id is used as an account for dumpers so that they can log in without having the root password. By placing them in the “operator” group, they can get read access to the disks. The “uucp” login has existed long before 4.3BSD, and is noted here just to provide a common user-id. The password entry “nobody” has been added to specify the user with least privilege.

After installing your updated password file, you must run `mkpasswd(8)` to create the *ndbm* password database. Note that `mkpasswd` is run whenever `vipw(8)` is run.

The format of the cron table, `/usr/lib/crontab`, has been changed to specify the user-id that should be used to run a process. The userid “nobody” is frequently useful for non-privileged programs.

Some of the commands previously in `/etc/rc.local` have been moved to `/etc/rc`; several new functions are now handled by `/etc/rc.local`. You should look closely at the prototype version of `/etc/rc.local` and read the manual pages for the commands contained in it before trying to merge your local copy. Note in particular that *ifconfig* has had many changes, and that host names are now fully

specified as domain-style names (e.g. `monet.Berkeley.EDU`) for the benefit of the name server.

The C library and system binaries on the distribution tape are compiled with new versions of *gethostbyname* and *gethostbyaddr* which use the name server, *named*(8). If you have only a small network and are not connected to a large network, you can use the distributed library routines without any problems; they use a linear scan of the host table */etc/hosts* if the name server is not running. If you are on the DARPA Internet or have a large local network, it is recommended that you set up and use the name server. For instructions on how to set up the necessary configuration files, refer to "Name Server Operations Guide for BIND". Several programs rely on the host name returned by *gethostname* to determine the local domain name.

If you want to compile your system to use the host table lookup routines instead of the name server, you will need to modify */usr/src/lib/libc/Makefile* according to the instructions there and then recompile all of the system and local programs (see section 6.6). Next, you must run *mkhosts*(8) to create the *ndbm* host table database from */etc/hosts*.

The format of */etc/ttys* has changed, see *ttys*(5) for details. It now includes the terminal type and security options that were previously placed in */etc/ttytype* and */etc/securettys*.

There is a new version of *syslog* that uses a more generalized facility/priority scheme. This has changed the format of the *syslog.conf* file. See *syslogd*(8) for details. *Syslog* now logs kernel errors, allowing events such as soft disk errors, filesystem-full messages, and other such error messages to be logged without slowing down the system while the messages print on the console. It is also used by many of the system daemons to monitor system problems more closely, for example network routing changes.

If you are using the name server, your *sendmail* configuration file will need some minor updates to accommodate it. See the "Sendmail Installation and Operation Guide" and the sample *sendmail* configuration files in */usr/src/usr.lib/sendmail/nscf*. Be sure to regenerate your sendmail frozen configuration file after installation of your updated configuration file.

The spooling directories saved on tape may be restored in their eventual resting places without too much concern. Be sure to use the 'p' option to *tar* so that files are recreated with the same file modes:

```
# cd /usr
# tar xp mgs spool/mail spool/uucp spool/uucppublic spool/news
```

The ownership and modes of two of these directories *at* now runs set-user-id "daemon" instead of root. Also, the *uucp* directory no longer needs to be publicly writable, as *tip* reverts to privileged status to remove its lock files. After copying your version of */usr/spool*, you should do:

```
# chown -R daemon /usr/spool/at
# chown -R root /usr/spool/uucp
# chgrp -R daemon /usr/spool/uucp
# chmod -R o-w /usr/spool/uucp
```

Whatever else is left is likely to be site specific or require careful scrutiny before placing in its eventual resting place. Refer to the documentation and source code before arbitrarily overwriting a file.

### 3.3. Hints on converting from 4.2BSD to 4.3BSD

This section summarizes the most significant changes between 4.2BSD and 4.3BSD, particularly those that are likely to cause difficulty in doing the conversion. It does not include changes in the network; see chapter 5 for information on setting up the network.

The mailbox locking protocol has changed; it now uses the advisory locking facility to avoid concurrent update of users' mail boxes. If you have your own mail interface, be sure to update its locking protocol.

The kernel's limit on the number of open files has been increased from 20 to 64. It is now possible to change this limit almost arbitrarily (there used to be a hard limit of 30). The standard I/O library autoconfigures to the kernel limit. Note that file ("\_job") entries may be allocated by *malloc* from *fopen*; this allocation has been known to cause problems with programs that use their own memory allocators. This does not occur until after 20 files have been opened by the standard I/O library.

*Select* can be used with more than 32 descriptors by using arrays of ints for the bit fields rather than single ints. Programs that used *getdtablesize* as their first argument to *select* will no longer work correctly. Usually the program can be modified to correctly specify the number of bits in an 'int. Alternatively the program can be modified to use an array of ints. There are a set of macros available in *<sys/types.h>* to simplify this. See *select(2)*.

Old core files will not be intelligible by the current debuggers because of numerous changes to the user structure and because the kernel stack has been enlarged. The *a.out* header that was in the user structure is no longer present. Locally-written debuggers that try to check the magic number will need modification.

*Find* now has a database of file names, constructed once a week from *cron*. To find a file by name only, the command *find name* will look in the database for files that match the name. This is much faster than *find / -name name -print*.

Files may not be deleted from directories having the "sticky" (ISVTX) bit set in their modes except by the owner of the file or of the directory, or by the superuser. This is primarily to protect users' files in publicly-writable directories such as */tmp* and */usr/tmp*. All publicly-writable directories should have their "sticky" bits set with "chmod +t."

The include file *<time.h>* has returned to */usr/include*, and again contains the definitions for the C library time routines of *ctime(3)*.

The *compact* and *uncompact* programs have been supplanted by the faster *compress*. If your user population has *compact*ed files, you will want to install *uncompact* found in */usr/src/old/compact*.

The configuration of the virtual memory limits has been simplified. A *MAXDSIZ* option, specified in bytes in the machine configuration file, may be used to raise the maximum process region size from the default of 17Mb to 32Mb or 64Mb. The initial per-process limit is still 6Mb, but can be raised up to *MAXDSIZ* with the *csd limit* command.

Some 4.3BSD binaries will not run with a 4.2BSD kernel because they take advantage of new functionality in 4.3BSD. One noticeable example of this problem is *csd*.

If you want to use *ps* after booting a new kernel, and before going multiuser, you must initialize its name list database by running *ps -U*.

## 4. SYSTEM SETUP

This section describes procedures used to set up a VAX UNIX system. These procedures are used when a system is first installed or when the system configuration changes. Procedures for normal system operation are described in the next section.

### 4.1. Creating UNIX boot media

The procedures for making the various UNIX boot media are described in this section. If you have an 11/785 or 11/780, you will need to make a floppy. For an 11/730, you will need to make a cassette. While for an 8650 or 8600, you will need to make a console RL02 pack.

The boot command files are all set up for booting off of the first UNIBUS or MASSBUS. If you are booting off of a different UNIBUS or MASSBUS, you will need to modify the boot command files appropriately.

#### 4.1.1. Making a UNIX boot console RL02 pack

If you have an 8650 or 8600 you will want to create a UNIX boot console RL02 pack by adding some files to your current DEC console pack, using *arff*(8). If you do not want to modify your current DEC console pack, you may make a copy of it first using *dd*(1). This pack will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console RL02 information is stored:

```
# cd /sys/console/rl
```

then set up the default boot device. If you have an RK07 as your primary root do:

```
# cp defboo.hk defboo.com
```

If you have a drive on a UDA50 (e.g. an RA81) as your primary root do:

```
# cp defboo.ra defboo.com
```

If you have a second vendor UNIBUS storage module as your primary root do:

```
# cp defboo.up defboo.com
```

Otherwise:

```
# cp defboo.hp defboo.com
```

The final step in updating the console RL02 pack is:

```
# make update
```

More copies of this console RL02 pack can be made using *dd*(1).

#### 4.1.2. Making a UNIX boot floppy

If you have an 11/785 or 11/780 you will want to create a UNIX boot floppy by adding some files to a copy of your current DEC console floppy, using *flcopy*(8) and *arff*(8). This floppy will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console floppy information is stored:

```
# cd /sys/floppy
```

then set up the default boot device. If you have an RK07 as your primary root do:

```
# cp defboo.hk defboo.cmd
```

If you have a drive on a UDA50 (e.g. an RA81) as your primary root do:

April 16, 1986

```
# cp defboo.ra defboo.cmd
```

If you have a second vendor UNIBUS storage module as your primary root do:

```
# cp defboo.up defboo.cmd
```

Otherwise:

```
# cp defboo.hp defboo.cmd
```

If the local configuration requires any changes in `restar.cmd` or `defboo.cmd` (e.g., for interleaved memory controllers see `defboo.MS780C-interleaved`), these should be made now. The following command will then copy your DEC local console floppy, updating the copy appropriately.

```
# make update
Change Floppy, Hit return when done.
(waits for you to put clean floppy in console)
Are you sure you want to clobber the floppy? yes
```

More copies of this floppy can be made using `flcopy(8)`.

#### 4.1.3. Making a UNIX boot cassette

If you have an 11/730 you will want to create a UNIX boot cassette by adding some files to a copy of your current DEC console cassette, using `flcopy(8)` and `arff(8)`. This cassette will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console cassette information is stored:

```
# cd /sys/cassette
```

then set up the default boot device. If you have an IDC storage module as your primary root do:

```
# cp defboo.rb defboo.cmd
```

If you have an RK07 as your primary root do:

```
# cp defboo.hk defboo.cmd
```

If you have a drive on a UDA50 as your primary root do:

```
# cp defboo.ra defboo.cmd
```

Otherwise:

```
# cp defboo.up defboo.cmd
```

To complete the procedure place your DEC local console cassette in drive 0 (the drive at front of the CPU); the following command will then copy it, updating the copy appropriately.

```
# make update
Change Floppy, Hit return when done.
(waits for you to put clean cassette in console drive 0)
Are you sure you want to clobber the floppy? yes
```

More copies of this cassette can best be made using `dd(1)`.

#### 4.2. Kernel configuration

This section briefly describes the layout of the kernel code and how files for devices are made. For a full discussion of configuring and building system images, consult the document "Building 4.3BSD UNIX Systems with Config".

April 16, 1986

#### 4.2.1. Kernel organization

As distributed, the kernel source is in a separate tar image. The source may be physically located anywhere within any file system so long as a symbolic link to the location is created for the file `/sys` (many files in `/usr/include` are normally symbolic links relative to `/sys`). In further discussions of the system source all path names will be given relative to `/sys`.

The directory `/sys/sys` contains the mainline machine independent operating system code. Files within this directory are conventionally named with the following prefixes:

<code>init_</code>	system initialization
<code>kern_</code>	kernel (authentication, process management, etc.)
<code>quota_</code>	disk quotas
<code>sys_</code>	system calls and similar
<code>tty_</code>	terminal handling
<code>ufs_</code>	file system
<code>uipc_</code>	interprocess communication
<code>vm_</code>	virtual memory

The remaining directories are organized as follows:

<code>/sys/h</code>	machine independent include files
<code>/sys/conf</code>	site configuration files and basic templates
<code>/sys/net</code>	network independent, but network related code
<code>/sys/netinet</code>	DARPA Internet code
<code>/sys/netimp</code>	IMP support code
<code>/sys/netns</code>	Xerox NS support code
<code>/sys/vax</code>	VAX specific mainline code
<code>/sys/vaxif</code>	VAX network interface code
<code>/sys/vaxmba</code>	VAX MASSBUS device drivers and related code
<code>/sys/vaxuba</code>	VAX UNIBUS device drivers and related code

Many of these directories are referenced through `/usr/include` with symbolic links. For example, `/usr/include/sys` is a symbolic link to `/sys/h`. The system code, as distributed, is totally independent of the include files in `/usr/include`. This allows the system to be recompiled from scratch without the `/usr` file system mounted.

#### 4.2.2. Devices and device drivers

Devices supported by UNIX are implemented in the kernel by drivers whose source is kept in `/sys/vax`, `/sys/vaxuba`, or `/sys/vaxmba`. These drivers are loaded into the system when included in a cpu specific configuration file kept in the `conf` directory. Devices are accessed through special files in the file system, made by the `mknod(8)` program and normally kept in the `/dev` directory. For all the devices supported by the distribution system, the files in `/dev` are created by the `/dev/MAKEDEV` shell script.

Determine the set of devices that you have and create a new `/dev` directory by running the `MAKEDEV` script. First create a new directory `/newdev`, copy `MAKEDEV` into it, edit the file `MAKEDEV.local` to provide an entry for local needs, and run it to generate a `/newdev` directory. For instance, if your machine has a single DZ11, a single DH11, a single DMF32, an RM03 disk, an EMULEX UNIBUS SMD disk controller, an AMPEX 9300 disk, and a TE16 tape drive you would do:

```
# cd /
# mkdir newdev
# cp dev/MAKEDEV newdev/MAKEDEV
# cd newdev
# MAKEDEV dz0 dh0 dm0 hp0 up0 ht0 std LOCAL
```

Note the "std" argument causes standard devices such as */dev/console*, the machine console, */dev/floppy*, the console floppy disk interface for the 11/780 and 11/785, and */dev/tu0* and */dev/tu1*, the console cassette interfaces for the 11/750 and 11/730, to be created.

You can then do

```
# cd /
# mv dev olddev ; mv newdev dev
# sync
```

to install the new device directory.

#### 4.2.3. Building new system images

The kernel configuration of each UNIX system is described by a single configuration file, stored in the */sys/conf* directory. To learn about the format of this file and the procedure used to build system images, start by reading "Building 4.3BSD UNIX Systems with Config", look at the manual pages in section 4 of the UNIX manual for the devices you have, and look at the sample configuration files in the */sys/conf* directory.

The configured system image "vmunix" should be copied to the root, and then booted to try it out. It is best to name it */newvmunix* so as not to destroy the working system until you're sure it does work:

```
# cp vmunix /newvmunix
# sync
```

It is also a good idea to keep the previous system around under some other name. In particular, we recommend that you save the generic distribution version of the system permanently as */genvmunix* for use in emergencies. To boot the new version of the system you should follow the bootstrap procedures outlined in section 6.1. After having booted and tested the new system, it should be installed as */vmunix* before going into multiuser operation. A systematic scheme for numbering and saving old versions of the system may be useful.

#### 4.3. Disk configuration

This section describes how to layout file systems to make use of the available space and to balance disk load for better system performance.

##### 4.3.1. Initializing */etc/fstab*

Change into the directory */etc* and copy the appropriate file from:

```
fstab.rm03
fstab.rm05
fstab.rm80
fstab.ra60
fstab.ra80
fstab.ra81
fstab.rb80
fstab.rp06
fstab.rp07
fstab.rk07
fstab.up160m (160Mb up drives)
fstab.hp400m (400Mb hp drives)
fstab.up (other up drives)
fstab.hp (other hp drives)
```

to the file `/etc/fstab`, i.e.:

```
# cd /etc
# cp fstab.xxx fstab
```

This will set up the default information about the usage of disk partitions, which we see how to update more below.

#### 4.3.2. Disk naming and divisions

Each physical disk drive can be divided into up to 8 partitions; UNIX typically uses only 3 or 4 partitions. For instance, on an RM03 or RP06, the first partition, `hp0a`, is used for a root file system, a backup thereof, or a small file system like, `/tmp`; the second partition, `hp0b`, is used for paging and swapping; and the third partition `hp0g` holds a user file system. On an RM05, the first three partitions are used as for the RM03, and the fourth partition, `hp0h`, holds the `/usr` file system, including source code.

The disk partition sizes for a drive are based on a set of four prototype partition tables; c.f. *diskpart*(8). The particular table used is dependent on the size of the drive. The "a" partition is the same size across all drives, 15884 sectors. The "b" partition, used for paging and swapping, is sized according to the total space on the disk. For drives less than about 400 megabytes the partition is 33440 sectors, while for larger drives the partition size is doubled to 66880 sectors. The "c" partition is always used to access the entire physical disk, including the space at the back of the disk reserved for the bad sector forwarding table. If the disk is larger than about 250 megabytes, an "h" partition is created with size 291346 sectors, and no matter whether the "h" partition is created or not, the remainder of the drive is allocated to the "g" partition. Sites that want to split up the "g" partition into several smaller file systems may use the "d", "e", and "f" partitions that overlap the "g" partition. The default sizes for these partitions are 15884, 55936, and the remainder of the disk, respectively\*.

The disk partition sizes for DEC RA60, RA80, and RA81 have changed since 4.2BSD. If upgrading from 4.2BSD, you will need to decide if you want to use the new partitions or the old partitions. If you desire to use the old partitions, you will need to update `/etc/disktab` and the device driver for the UDA50. Any other partition sizes that were modified at your site, will require the same consideration.

The space available on a disk varies per device. The amount of space available on the common disk partitions is listed in the following table. Not shown in the table are the partitions of each drive devoted to the root file system and the paging area.

\* These rules are, unfortunately not evenly applied to all disks. Drives on DEC UDA50 and IDC controllers do not completely follow these rules; in particular, no "d", "e", or "f" partitions are available on an RA60 and RA80. Consult *uda*(4) for more information.



Type	Name	Size	Name	Size
rk07	hk?g	13 Mb		
rm03	hp?g	41 Mb		
rp06	hp?g	145 Mb		
rm05	hp?g	80 Mb	hp?h	145 Mb
rm80	hp?g	96 Mb		
ra60	ra?g	78 Mb	ra?h	96 Mb
ra80	ra?g	96 Mb		
ra81	ra?g	257 Mb	ra?h	145 Mb
rb80	rb?g	41 Mb	rb?h	56 Mb
rp07	hp?g	315 Mb	hp?h	145 Mb
up300	up?g	80 Mb	up?h	145 Mb
up330	up?g	90 Mb	up?h	145 Mb
up400	hp?g	216 Mb	hp?h	145 Mb
up160	up?g	106 Mb		

Here up300 refers to either an AMPEX or CDC 300 Megabyte disk on a MASSBUS or UNIBUS disk controller, up330 refers to either an AMPEX or FUJITSU 330 Megabyte disk on a MASSBUS or UNIBUS controller, up160 refers to a FUJITSU 160 Megabyte disk on the UNIBUS, and up400 refers to a FUJITSU Eagle 400 Megabyte disk on a MASSBUS or UNIBUS disk controller. "hp" should be substituted for "up" above if the disk is on the MASSBUS. Consult the manual pages for the specific controllers for other supported disks or other partitions.

Each disk also has a paging area, typically of 16 Megabytes, and a root file system of 8 Megabytes. The distributed system binaries occupy about 34 Megabytes while the major sources occupy another 32 Megabytes. This overflows dual RK07, dual RL02 and single RM03 systems, but fits easily on most other hardware configurations.

Be aware that the disks have their sizes measured in disk sectors (512 bytes), while the UNIX file system blocks are variable sized. All user programs report disk space in kilobytes and, where needed, disk sizes are always specified in units of sectors. The `/etc/disktab` file used in making file systems specifies disk partition sizes in sectors; the default sector size may be overridden with the "se" attribute. Note that the only sector size currently supported is `DEV_BSIZE` as defined in `/sys/h/param.h`.

#### 4.3.3. Layout considerations

There are several considerations in deciding how to adjust the arrangement of things on your disks. The most important is making sure that there is adequate space for what is required; secondarily, throughput should be maximized. Paging space is an important parameter. The system, as distributed, sizes the configured paging areas each time the system is booted. Further, multiple paging areas of different size may be interleaved. Drives smaller than 400 megabytes have swap partitions of 16 megabytes while drives larger than 400 megabytes have 32 megabytes. These values may be changed to get more paging space by changing the appropriate partition table in the disk driver.

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the `/tmp` directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks; if you have several disks, it makes sense to mount this in a "root" (i.e. first partition) file system on another disk. All the programs that create files in `/tmp` take care to delete them, but are not immune to rare events and can leave dregs. The directory should be examined every so often and the old files deleted.

The efficiency with which UNIX is able to use the CPU is often strongly affected by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split the root file system (`/`), system binaries (`/usr`), the temporary files (`/tmp`), and the user files among several disk arms, and to interleave the paging activity among several arms.

It is critical for good performance to balance disk load. There are at least five components of the disk load that you can divide between the available disks:

1. The root file system.
2. The /tmp file system.
3. The /usr file system.
4. The user files.
5. The paging activity.

The following possibilities are ones we have used at times when we had 2, 3 and 4 disks:

what	disks		
	2	3	4
/	0	0	0
tmp	1	2	3
usr	1	1	1
paging	0+1	0+2	0+2+3
users	0	0+2	0+2
archive	x	x	3

The most important things to consider are to even out the disk load as much as possible, and to do this by decoupling file systems (on separate arms) between which heavy copying occurs. Note that a long term average balanced load is not important; it is much more important to have an instantaneously balanced load when the system is busy.

Intelligent experimentation with a few file system arrangements can pay off in much improved performance. It is particularly easy to move the root, the /tmp file system and the paging areas. Place the user files and the /usr directory as space needs dictate and experiment with the other, more easily moved file systems.

#### 4.3.4. File system parameters

Each file system is parameterized according to its block size, fragment size, and the disk geometry characteristics of the medium on which it resides. Inaccurate specification of the disk characteristics or haphazard choice of the file system parameters can result in substantial throughput degradation or significant waste of disk space. As distributed, file systems are configured according to the following table.

File system	Block size	Fragment size
/	8 Kbytes	1 Kbytes
usr	4 Kbytes	512 bytes
users	4 Kbytes	1 Kbytes

The root file system block size is made large to optimize bandwidth to the associated disk; this is particularly important since the /tmp directory is normally part of the root file or a similar filesystem. The large block size is also important as many of the most heavily used programs are demand paged out of the /bin directory. The fragment size of 1 Kbytes is a "nominal" value to use with a file system. With a 1 Kbyte fragment size disk space utilization is about the same as with the earlier versions of the file system.

The usr file system uses a 4 Kbyte block size with 512 byte fragment size in an effort to get high performance while conserving the amount of space wasted by a large fragment size. Space compaction has been deemed important here because the source code for the system is normally placed on this file system. If the source code is placed on a separate filesystem, use of an 8 Kbyte block size with 1 Kbyte fragments might be considered for improved performance when paging from /usr binaries.

The file systems for users have a 4 Kbyte block size with 1 Kbyte fragment size. These parameters have been selected based on observations of the performance of our user file systems. The 4 Kbyte block size provides adequate bandwidth while the 1 Kbyte fragment size provides acceptable space compaction and disk fragmentation.

Other parameters may be chosen in constructing file systems, but the factors involved in choosing a block size and fragment size are many and interact in complex ways. Larger block sizes result in better throughput to large files in the file system as larger I/O requests will then be performed by the system. However, consideration must be given to the average file sizes found in the file system and the performance of the internal system buffer cache. The system currently provides space in the inode for 12 direct block pointers, 1 single indirect block pointer, and 1 double indirect block pointer.\* If a file uses only direct blocks, access time to it will be optimized by maximizing the block size. If a file spills over into an indirect block, increasing the block size of the file system may decrease the amount of space used by eliminating the need to allocate an indirect block. However, if the block size is increased and an indirect block is still required, then more disk space will be used by the file because indirect blocks are allocated according to the block size of the file system.

In selecting a fragment size for a file system, at least two considerations should be given. The major performance tradeoffs observed are between an 8 Kbyte block file system and a 4 Kbyte block file system. Because of implementation constraints, the block size / fragment size ratio can not be greater than 8. This means that an 8 Kbyte file system will always have a fragment size of at least 1 Kbytes. If a file system is created with a 4 Kbyte block size and a 1 Kbyte fragment size, then upgraded to an 8 Kbyte block size and 1 Kbyte fragment size, identical space compaction will be observed. However, if a file system has a 4 Kbyte block size and 512 byte fragment size, converting it to an 8K/1K file system will result in significantly more space being used. This implies that 4 Kbyte block file systems that might be upgraded to 8 Kbyte blocks for higher performance should use fragment sizes of at least 1 Kbytes to minimize the amount of work required in conversion.

A second, more important, consideration when selecting the fragment size for a file system is the level of fragmentation on the disk. With a 512 byte fragment size, storage fragmentation occurs much sooner, particularly with a busy file system running near full capacity. By comparison, the level of fragmentation in a 1 Kbyte fragment file system is one tenth as severe. This means that on file systems where many files are created and deleted, the 512 byte fragment size is more likely to result in apparent space exhaustion because of fragmentation. That is, when the file system is nearly full, file expansion that requires locating a contiguous area of disk space is more likely to fail on a 512 byte file system than on a 1 Kbyte file system. To minimize fragmentation problems of this sort, a parameter in the super block specifies a minimum acceptable free space threshold. When normal users (i.e. anyone but the super-user) attempt to allocate disk space and the free space threshold is exceeded, the user is returned an error as if the file system were really full. This parameter is nominally set to 10%; it may be changed by supplying a parameter to *newfs*, or by updating the super block of an existing file system using *tune2fs(8)*.

In general, unless a file system is to be used for a special purpose application (for example, storing image processing data), we recommend using the values supplied above. Remember that the current implementation limits the block size to at most 8 Kbytes and the ratio of block size / fragment size must be 1, 2, 4, or 8.

The disk geometry information used by the file system affects the block layout policies employed. The file */etc/disktab*, as supplied, contains the data for most all drives supported by the system. When constructing a file system you should use the *newfs(8)* program and specify the type of disk on which the file system resides. This file also contains the default file system partition sizes, and default block and fragment sizes. To override any of the default values you can modify the file or use an option to *newfs*.

---

\* A triple indirect block pointer is also reserved, but not currently supported.

#### 4.3.5. Implementing a layout

To put a chosen disk layout into effect, you should use the `newfs(8)` command to create each new file system. Each file system must also be added to the file `/etc/fstab` so that it will be checked and mounted when the system is bootstrapped.

As an example, consider a system with RM80's. On the first RM80, `hp0`, we will put the root file system in `hp0a`, and the `/usr` file system in `hp0g`, which has enough space to hold it and then some. The `/tmp` directory will be part of the root file system, as no file system will be mounted on `/tmp`. If we had only one RM80, we would put user files in the `hp0g` partition with the system source and binaries.

If we had a second RM80, we would place `/usr` in `hp1g`. We would put user files in `hp0g`, calling the file system `/mnt`. We would also interleave the paging between the 2 RM80's. To do this we would build a system configuration that specified:

```
config    vmunix    root on hp0 swap on hp0 and hp1
```

to get the swap interleaved, and `/etc/fstab` would then contain

```
/dev/hp0a:rw:1:1
/dev/hp0b::sw::
/dev/hp0g:/mnt:rw:1:2
/dev/hp1b::sw::
/dev/hp1g:/usr:rw:1:2
```

*We would keep a backup copy of the root file system in the `hp1a` disk partition. Alternatively, that partition could be used for `/tmp`.*

To make the `/mnt` file system we would do:

```
# cd /dev
# MAKEDEV hp1
# newfs hp1g rm80
(information about file system prints out)
# mkdir /mnt
# mount /dev/hp1g /mnt
```

#### 4.4. Configuring terminals

If UNIX is to support simultaneous access from directly-connected terminals other than the console, the file `/etc/ttys` (`ttys(5)`) must be edited.

Terminals connected via DZ11 interfaces are conventionally named `ttyDD` where `DD` is a decimal number, the "minor device" number. The lines on `dz0` are named `/dev/tty00`, `/dev/tty01`, ... `/dev/tty07`. By convention, all other terminal names are of the form `ttyCX`, where `C` is an alphabetic character according to the type of terminal multiplexor and its unit number, and `X` is a digit for the first ten lines on the interface and an increasing lower case letter for the rest of the lines. `C` is defined for the number of interfaces of each type listed below.

Interface Type	Characters	Number of lines per board	Number of Interfaces
DZ11	see above	8	10
DMF32	A-C,E-I	8	8
DMZ32	a-c,e-g	24	6
DH11	h-o	16	8
DHU11	S-Z	16	8
pty	p-u	16	6

April 16, 1986

To add a new terminal device, be sure the device is configured into the system and that the special files for the device have been made by `/dev/MAKEDEV`. Then, enable the appropriate lines of `/etc/ttys` by setting the "status" field to `on` (or add new lines). Note that lines in `/etc/ttys` are one-for-one with entries in the file of current users (`/etc/utmp`), and therefore it is best to make changes while running in single-user mode and to add all of the entries for a new device at once.

The format of the `/etc/ttys` file is completely new in 4.3BSD. Each line in the file is broken into four tab separated fields (comments are shown by a `#` character and extend to the end of the line). For each terminal line the four fields are: the device (without a leading `/dev`), the program `/etc/init` should startup to service the line (or `none` if the line is to be left alone), the terminal type (found in `/etc/termcap`), and optional status information describing if the terminal is enabled or not and if it is "secure" (i.e. the super user should be allowed to login on the line). All fields are character strings with entries requiring embedded white space enclosed in double quotes. Thus a newly added terminal `/dev/tty00` could be added as

```
tty00      "/etc/getty std.9600" vt100      on secure # mike's office
```

The `std.9600` parameter provided to `/etc/getty` is used in searching the file `/etc/gettytab`; it specifies a terminal's characteristics (such as baud rate). To make custom terminal types, consult `gettytab(5)` before modifying `/etc/gettytab`.

Dialup terminals should be wired so that carrier is asserted only when the phone line is dialed up. For non-dialup terminals from which modem control is not available, you must either wire back the signals so that the carrier appears to always be present, or show in the system configuration that carrier is to be assumed to be present with *flags* for each terminal device. See *dh(4)*, *dhu(4)*, *dz(4)*, *dmz(4)*, and *dmf(4)* for details.

For network terminals (i.e. pseudo terminals), no program should be started up on the lines. Thus, the normal entry in `/etc/ttys` would look like

```
ttyp0      none network
```

(Note the fourth field is not needed when here.)

When the system is running multi-user, all terminals that are listed in `/etc/ttys` as `on` have their line enabled. If, during normal operations, it is desired to disable a terminal line, you can edit the file `/etc/ttys` to change the terminal's status to `off` and then send a hangup signal to the `init` process, by doing

```
# kill -1 1
```

Terminals can similarly be enabled by changing the status field from `off` to `on` and sending a hangup signal to `init`.

Note that if a special file is inaccessible when `init` tries to create a process for it, `init` will log a message to the system error logging process (`/etc/syslogd`) and try to reopen the terminal every minute, reprinting the warning message every 10 minutes. Messages of this sort are normally printed on the console, though other actions may occur depending on the configuration information found in `/etc/syslog.conf`.

Finally note that you should change the names of any dialup terminals to `ttyd?` where `?` is in `[0-9a-zA-Z]`, as some programs use this property of the names to determine if a terminal is a dialup. Shell commands to do this should be put in the `/dev/MAKEDEV.local` script.

While it is possible to use truly arbitrary strings for terminal names, the accounting and noticeably the `ps(1)` command make good use of the convention that `tty` names (by default, and also after dialups are named as suggested above) are distinct in the last 2 characters. Change this and you may be sorry later, as the heuristic `ps(1)` uses based on these conventions will then break down and `ps` will run MUCH slower.

#### 4.5. Adding users

New users can be added to the system by adding a line to the password file `/etc/passwd`. The procedure for adding a new user is described in *adduser*(8).

You should add accounts for the initial user community, giving each a directory and a password, and putting users who will wish to share software in the same groups.

Several guest accounts have been provided on the distribution system; these accounts are for people at Berkeley, Bell Laboratories, and others who have done major work on UNIX in the past. You can delete these accounts, or leave them on the system if you expect that these people would have occasion to login as guests on your system.

#### 4.6. Site tailoring

All programs that require the site's name, or some similar characteristic, obtain the information through system calls or from files located in `/etc`. Aside from parts of the system related to the network, to tailor the system to your site you must simply select a site name, then edit the file

`/etc/rc.local`

The first line in `/etc/rc.local`,

`/bin/hostname mysitename`

defines the value returned by the *gethostname*(2) system call. If you are running the name server, your site name should be your fully qualified domain name. Programs such as *getty*(8), *mail*(1), *wall*(1), *uucp*(1), and *who*(1) use this system call so that the binary images are site independent.

#### 4.7. Setting up the line printer system

The line printer system consists of at least the following files and commands:

<code>/usr/ucb/lpq</code>	spooling queue examination program
<code>/usr/ucb/lprm</code>	program to delete jobs from a queue
<code>/usr/ucb/lpr</code>	program to enter a job in a printer queue
<code>/etc/printcap</code>	printer configuration and capability data base
<code>/usr/lib/lpd</code>	line printer daemon, scans spooling queues
<code>/etc/lpc</code>	line printer control program
<code>/etc/hosts.lpd</code>	list of host allowed to use the printers

The file `/etc/printcap` is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page *printcap*(5) describes the format of this data base and also shows the default values for such things as the directory in which spooling is performed. The line printer system handles multiple printers, multiple spooling queues, local and remote printers, and also printers attached via serial lines that require line initialization such as the baud rate. Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

Remote spooling via the network is handled with two spooling queues, one on the local machine and one on the remote machine. When a remote printer job is started with *lpr*, the job is queued locally and a daemon process created to oversee the transfer of the job to the remote machine. If the destination machine is unreachable, the job will remain queued until it is possible to transfer the files to the spooling queue on the remote machine. The *lpq* program shows the contents of spool queues on both the local and remote machines.

To configure your line printers, consult the *printcap* manual page and the accompanying document, "4.3BSD Line Printer Spooler Manual". A call to the *lpd* program should be present in `/etc/rc`.

#### 4.8. Setting up the mail system

The mail system consists of the following commands:

/bin/mail	old standard mail program, <i>binmail</i> (1)
/usr/ucb/mail	UCB mail program, described in <i>mail</i> (1)
/usr/lib/sendmail	mail routing program
/usr/spool/mail	mail spooling directory
/usr/spool/secretmail	secure mail directory
/usr/bin/xsend	secure mail sender
/usr/bin/xget	secure mail receiver
/usr/lib/aliases	mail forwarding information
/usr/ucb/newaliases	command to rebuild binary forwarding database
/usr/ucb/biff	mail notification enabler
/etc/comsat	mail notification daemon

Mail is normally sent and received using the *mail*(1) command, which provides a front-end to edit the messages sent and received, and passes the messages to *sendmail*(8) for routing. The routing algorithm uses knowledge of the network name syntax, aliasing and forwarding information, and network topology, as defined in the configuration file */usr/lib/sendmail.cf*, to process each piece of mail. Local mail is delivered by giving it to the program */bin/mail* that adds it to the mailboxes in the directory */usr/spool/mail/username*, using a locking protocol to avoid problems with simultaneous updates. After the mail is delivered, the local mail delivery daemon */etc/comsat* is notified, which in turn notifies users who have issued a "*biff*" command that mail has arrived.

Mail queued in the directory */usr/spool/mail* is normally readable only by the recipient. To send mail that is secure against any possible perusal (except by a code-breaker) you should use the secret mail facility, which encrypts the mail so that no one can read it.

To set up the mail facility you should read the instructions in the file *READ\_ME* in the directory */usr/src/usr.lib/sendmail* and then adjust the necessary configuration files. You should also set up the file */usr/lib/aliases* for your installation, creating mail groups as appropriate. Documents describing *sendmail*'s operation and installation are also included in the distribution.

##### 4.8.1. Setting up a UUCP connection

The version of *uucp* included in 4.3BSD is an enhanced version of the one originally distributed with 32/V\*. The enhancements include:

- support for many auto call units and dialers in addition to the DEC DN11,
- breakup of the spooling area into multiple subdirectories,
- addition of an *L.cmds* file to control the set of commands that may be executed by a remote site,
- enhanced "expect-send" sequence capabilities when logging in to a remote site,
- new commands to be used in polling sites and obtaining snap shots of *uucp* activity,
- additional protocols for different communication media.

This section gives a brief overview of *uucp* and points out the most important steps in its installation.

To connect two UNIX machines with a *uucp* network link using modems, one site must have an automatic call unit and the other must have a dialup port. It is better if both sites have both.

You should first read the paper in the UNIX System Manager's Manual: "Uucp Implementation Description". It describes in detail the file formats and conventions, and will give you a little context. In addition, the document "setup.tblms", located in the directory */usr/src/usr.bin/uucp/UUAIDS*, may be of use in tailoring the software to your needs.

\* The *uucp* included in this distribution is the result of work by many people; we gratefully acknowledge their contributions, but refrain from mentioning names in the interest of keeping this document current.

The *uucp* support is located in three major directories: */usr/bin*, */usr/lib/uucp*, and */usr/spool/uucp*. User commands are kept in */usr/bin*, operational commands in */usr/lib/uucp*, and */usr/spool/uucp* is used as a spooling area. The commands in */usr/bin* are:

<i>/usr/bin/uucp</i>	file-copy command
<i>/usr/bin/uux</i>	remote execution command
<i>/usr/bin/uusend</i>	binary file transfer using mail
<i>/usr/bin/uuencode</i>	binary file encoder (for <i>uusend</i> )
<i>/usr/bin/uudecode</i>	binary file decoder (for <i>uusend</i> )
<i>/usr/bin/uulog</i>	scans session log files
<i>/usr/bin/uusnap</i>	gives a snap-shot of <i>uucp</i> activity
<i>/usr/bin/uupoll</i>	polls remote system until an answer is received
<i>/usr/bin/uuname</i>	prints a list of known <i>uucp</i> hosts
<i>/usr/bin/uuq</i>	gives information about the queue

The important files and commands in */usr/lib/uucp* are:

<i>/usr/lib/uucp/L-devices</i>	list of dialers and hard-wired lines
<i>/usr/lib/uucp/L-dialcodes</i>	dialcode abbreviations
<i>/usr/lib/uucp/L.aliases</i>	hostname aliases
<i>/usr/lib/uucp/L.cmds</i>	commands remote sites may execute
<i>/usr/lib/uucp/L.sys</i>	systems to communicate with, how to connect, and when
<i>/usr/lib/uucp/SEQF</i>	sequence numbering control file
<i>/usr/lib/uucp/USERFILE</i>	remote site pathname access specifications
<i>/usr/lib/uucp/uucico</i>	<i>uucp</i> protocol daemon
<i>/usr/lib/uucp/uuclean</i>	cleans up garbage files in spool area
<i>/usr/lib/uucp/uuxqt</i>	<i>uucp</i> remote execution server

while the spooling area contains the following important files and directories:

<i>/usr/spool/uucp/C.</i>	directory for command, "C." files
<i>/usr/spool/uucp/D.</i>	directory for data, "D.", files
<i>/usr/spool/uucp/X.</i>	directory for command execution, "X.", files
<i>/usr/spool/uucp/D.machine</i>	directory for local "D." files
<i>/usr/spool/uucp/D.machineX</i>	directory for local "X." files
<i>/usr/spool/uucp/TM.</i>	directory for temporary, "TM.", files
<i>/usr/spool/uucp/LOGFILE</i>	log file of <i>uucp</i> activity
<i>/usr/spool/uucp/SYSLOG</i>	log file of <i>uucp</i> file transfers

To install *uucp* on your system, start by selecting a site name (shorter than 14 characters). A *uucp* account must be created in the password file and a password set up. Then, create the appropriate spooling directories with mode 755 and owned by user *uucp*, group *daemon*.

If you have an auto-call unit, the *L.sys*, *L-dialcodes*, and *L-devices* files should be created. The *L.sys* file should contain the phone numbers and login sequences required to establish a connection with a *uucp* daemon on another machine. For example, our *L.sys* file looks something like:

```
adiron Any ACU 1200 out0123456789- ogin-EOT-ogin uucp
cbosg Never Slave 300
cbosgd Never Slave 300
chico Never Slave 1200 out2010123456
```

The first field is the name of a site, the second shows when the machine may be called, the third field specifies how the host is connected (through an ACU, a hard-wired line, etc.), then comes the phone number to use in connecting through an auto-call unit, and finally a login sequence. The phone number may contain common abbreviations that are defined in the *L-dialcodes* file. The device



specification should refer to devices specified in the L-devices file. Listing only ACU causes the *uucp* daemon, *uucico*, to search for any available auto-call unit in L-devices. Our L-dialcodes file is of the form:

```
ucb 2
out 9%
```

while our L-devices file is:

```
ACU cul0 unused 1200 ventel
```

Refer to the README file in the *uucp* source directory for more information about installation.

As *uucp* operates it creates (and removes) many small files in the directories underneath /usr/spool/uucp. Sometimes files are left undeleted; these are most easily purged with the *uuclean* program. The log files can grow without bound unless trimmed back; *uulog* maintains these files. Many useful aids in maintaining your *uucp* installation are included in a subdirectory UUAIDS beneath /usr/src/usr.bin/uucp. Peruse this directory and read the "setup" instructions also located there.

April 16, 1986

## 5. NETWORK SETUP

4.3BSD provides support for the DARPA standard Internet protocols IP, ICMP, TCP, and UDP. These protocols may be used on top of a variety of hardware devices ranging from the IMP's (PSN's) used in the ARPANET to local area network controllers for the Ethernet. Network services are split between the kernel (communication protocols) and user programs (user services such as TELNET and FTP). This section describes how to configure your system to use the Internet networking support. 4.3BSD also supports the Xerox Network Systems (NS) protocols. IDP and SPP are implemented in the kernel, and other protocols such as Courier run at the user level.

### 5.1. System configuration

To configure the kernel to include the Internet communication protocols, define the INET option. Xerox NS support is enabled with the NS option. In either case, include the pseudo-devices "pty", and "loop" in your machine's configuration file. The "pty" pseudo-device forces the pseudo-terminal device driver to be configured into the system, see *pty(4)*, while the "loop" pseudo-device forces inclusion of the software loopback interface driver. The loop driver is used in network testing and also by the error logging system.

If you are planning to use the Internet network facilities on a 10Mb/s Ethernet, the pseudo-device "ether" should also be included in the configuration; this forces inclusion of the Address Resolution Protocol module used in mapping between 48-bit Ethernet and 32-bit Internet addresses. Also, if you have an IMP connection, you will need to include the pseudo-device "imp."

Before configuring the appropriate networking hardware, you should consult the manual pages in section 4 of the Programmer's Manual. The following table lists the devices for which software support exists.

Device name	Manufacturer and product
acc	ACC LH/DH interface to IMP
css	DEC IMP-11A interface to IMP
ddn	ACC ACP625 DDN Standard mode X.25 interface to IMP
dmc	DEC DMC-11 (also works with DMR-11)
de	DEC DEUNA 10Mb/s Ethernet
ec	3Com 10Mb/s Ethernet
en	Xerox 3Mb/s prototype Ethernet (not a product)
ex	Excelan 204 10Mb/s Ethernet
hdh	ACC IF-11/HDH IMP interface
hy	NSC Hyperchannel, w/ DR-11B and PI-13 interfaces
il	Interlan 1010 and 10101A 10Mb/s Ethernet interfaces
ix	Interlan NP100 10Mb/s Ethernet interface
pcl	DEC PCL-11
vv	Proteon 10Mb/s and 80Mb/s proNET ring network (V2LNI)

All network interface drivers including the loopback interface, require that their host address(es) be defined at boot time. This is done with *ifconfig(8C)* commands included in the */etc/rc.local* file. Interfaces that are able to dynamically deduce the host part of an address may check that the host part of the address is correct. The manual page for each network interface describes the method used to establish a host's address. *Ifconfig(8)* can also be used to set options for the interface at boot time. Options are set independently for each interface, and apply to all packets sent using that interface. These options include disabling the use of the Address Resolution Protocol; this may be useful if a network is shared with hosts running software that does not yet provide this function. Alternatively, translations for such hosts may be set in advance or "published" by a 4.3BSD host by use of the *arp(8c)* command. Note that the use of trailer link-level is now negotiated between 4.3BSD hosts using ARP, and it is thus no longer necessary to disable the use of trailers with *ifconfig*.

April 16, 1986

To use the pseudo terminals just configured, device entries must be created in the `/dev` directory. To create 32 pseudo terminals (plenty, unless you have a heavy network load) execute the following commands.

```
# cd /dev
# MAKEDEV pty0 pty1
```

More pseudo terminals may be made by specifying `pty2`, `pty3`, etc. The kernel normally includes support for 32 pseudo terminals unless the configuration file specifies a different number. Each pseudo terminal really consists of two files in `/dev`: a master and a slave. The master pseudo terminal file is named `/dev/ptyp?`, while the slave side is `/dev/ttyp?`. Pseudo terminals are also used by several programs not related to the network. In addition to creating the pseudo terminals, be sure to install them in the `/etc/tty` file (with a 'none' in the second column so no `getty` is started).

## 5.2. Local subnetworks

In 4.3BSD the DARPA Internet support includes the notion of "subnetworks". This is a mechanism by which multiple local networks may appear as a single Internet network to off-site hosts. Subnetworks are useful because they allow a site to hide their local topology, requiring only a single route in external gateways; it also means that local network numbers may be locally administered. The standard describing this change in Internet addressing is RFC-950.

To set up local subnetworks one must first decide how the available address space (the Internet "host part" of the 32-bit address) is to be partitioned. Sites with a class A network number have a 24-bit address space with which to work, sites with a class B network number have a 16-bit address space, while sites with a class C network number have an 8-bit address space\*. To define local subnets you must steal some bits from the local host address space for use in extending the network portion of the Internet address. This reinterpretation of Internet addresses is done only for local networks; i.e. it is not visible to hosts off-site. For example, if your site has a class B network number, hosts on this network have an Internet address that contains the network number, 16 bits, and the host number, another 16 bits. To define 254 local subnets, each possessing at most 255 hosts, 8 bits may be taken from the local part. (The use of subnets 0 and all-1's, 255 in this example, is discouraged to avoid confusion about broadcast addresses.) These new network numbers are then constructed by concatenating the original 16-bit network number with the extra 8 bits containing the local subnetwork number.

The existence of local subnetworks is communicated to the system at the time a network interface is configured with the `netmask` option to the `ifconfig` program. A "network mask" is specified to define the portion of the Internet address that is to be considered the network part for that network. This mask normally contains the bits corresponding to the standard network part as well as the portion of the local part that has been assigned to subnets. If no mask is specified when the address is set, it will be set according to the class of the network. For example, at Berkeley (class B network 128.32) 8 bits of the local part have been reserved for defining subnetworks; consequently the `/etc/rc.local` file contains lines of the form

```
/etc/ifconfig en0 netmask 0xffff00 128.32.1.7
```

This specifies that for interface "en0", the upper 24 bits of the Internet address should be used in calculating network numbers (netmask 0xffff00), and the interface's Internet address is "128.32.1.7" (host 7 on network 128.32.1). Hosts *m* on sub-network *n* of this network would then have addresses of the form "128.32.*n.m*"; for example, host 99 on network 129 would have an address "128.32.129.99". For hosts with multiple interfaces, the network mask should be set for each interface, although in practice only the mask of the first interface on each network is actually used.

\* If you are unfamiliar with the Internet addressing structure, consult "Address Mappings", Internet RFC-796, J. Postel; available from the Internet Network Information Center at SRI.

### 5.3. Internet broadcast addresses

The address defined as the broadcast address for Internet networks according to RFC-919 is the address with a host part of all 1's. The address used by 4.2BSD was the address with a host part of 0. 4.3BSD uses the standard broadcast address (all 1's) by default, but allows the broadcast address to be set (with *ifconfig*) for each interface. This allows networks consisting of both 4.2BSD and 4.3BSD hosts to coexist while the upgrade process proceeds. In the presence of subnets, the broadcast address uses the subnet field as for normal host addresses, with the remaining host part set to 1's (or 0's, on a network that has not yet been converted). 4.3BSD hosts recognize and accept packets sent to the logical-network broadcast address as well as those sent to the subnet broadcast address, and when using an all-1's broadcast, also recognize and receive packets sent to host 0 as a broadcast.

### 5.4. Routing

If your environment allows access to networks not directly attached to your host you will need to set up routing information to allow packets to be properly routed. Two schemes are supported by the system. The first scheme employs the routing table management daemon */etc/routed* to maintain the system routing tables. The routing daemon uses a variant of the Xerox Routing Information Protocol to maintain up to date routing tables in a cluster of local area networks. By using the */etc/gateways* file created by *htable(8)*, the routing daemon can also be used to initialize static routes to distant networks (see the next section for further discussion). When the routing daemon is started up (usually from */etc/rc.local*) it reads */etc/gateways* if it exists and installs those routes defined there, then broadcasts on each local network to which the host is attached to find other instances of the routing daemon. If any responses are received, the routing daemons cooperate in maintaining a globally consistent view of routing in the local environment. This view can be extended to include remote sites also running the routing daemon by setting up suitable entries in */etc/gateways*; consult *routed(8C)* for a more thorough discussion.

The second approach is to define a default or wildcard route to a smart gateway and depend on the gateway to provide ICMP routing redirect information to dynamically create a routing data base. This is done by adding an entry of the form

```
/etc/route add default smart-gateway 1
```

to */etc/rc.local*; see *route(8C)* for more information. The default route will be used by the system as a "last resort" in routing packets to their destination. Assuming the gateway to which packets are directed is able to generate the proper routing redirect messages, the system will then add routing table entries based on the information supplied. This approach has certain advantages over the routing daemon, but is unsuitable in an environment where there are only bridges (i.e. pseudo gateways that, for instance, do not generate routing redirect messages). Further, if the smart gateway goes down there is no alternative, save manual alteration of the routing table entry, to maintaining service.

The system always listens, and processes, routing redirect information, so it is possible to combine both of the above facilities. For example, the routing table management process might be used to maintain up to date information about routes to geographically local networks, while employing the wildcard routing techniques for "distant" networks. The *netstat(1)* program may be used to display routing table contents as well as various routing oriented statistics. For example,

```
# netstat -r
```

will display the contents of the routing tables, while

```
# netstat -r -s
```

will show the number of routing table entries dynamically created as a result of routing redirect messages, etc.

### 5.5. Use of 4.3BSD machines as gateways

Several changes have been made in 4.3BSD in the area of gateway support (or packet forwarding, if one prefers). A new configuration option, GATEWAY, is used when configuring a machine to be used as a gateway. This option increases the size of the routing hash tables in the kernel. Unless configured with that option, hosts with only a single non-loopback interface never attempt to forward packets or to respond with ICMP error messages to misdirected packets. This change reduces the problems that may occur when different hosts on a network disagree as to the network number or broadcast address. Another change is that 4.3BSD machines that forward packets back through the same interface on which they arrived will send ICMP redirects to the source host if it is on the same network. This improves the interaction of 4.3BSD gateways with hosts that configure their routes via default gateways and redirects. The generation of redirects may be disabled with the configuration option IPSENDREDIRECTS=0 in environments where it may cause difficulties.

Local area routing within a group of interconnected Ethernets and other such networks may be handled by *routed*(8c). Gateways between the Arpanet or Milnet and one or more local networks require an additional routing protocol, the Exterior Gateway Protocol (EGP), to inform the core gateways of their presence and to acquire routing information from the core. An EGP implementation for 4.2BSD was done by Paul Kirton while visiting ISI, and any sites requiring such support that have not already obtained a copy should contact Joyce Reynolds (JKReynolds@usc-isif.arpa) for information. That implementation works with 4.3BSD without kernel modifications. It must be modified, as packets from the ICMP raw socket include the IP header like other raw sockets in 4.3BSD. If necessary, contact the Berkeley Computer Systems Research Group for assistance.

### 5.6. Network servers

In 4.3BSD most of the server programs are started up by a "super server", the Internet daemon. The Internet daemon, */etc/inetd*, acts as a master server for programs specified in its configuration file, */etc/inetd.conf*, listening for service requests for these servers, and starting up the appropriate program whenever a request is received. The configuration file contains lines containing a service name (as found in */etc/services*), the type of socket the server expects (e.g. stream or dgram), the protocol to be used with the socket (as found in */etc/protocols*), whether to wait for each server to complete before starting up another, the user name as which the server should run, the server program's name, and at most five arguments to pass to the server program. Some trivial services are implemented internally in *inetd*, and their servers are listed as "internal." For example, an entry for the file transfer protocol server would appear as

```
ftp stream tcp nowait root /etc/ftpd ftpd
```

Consult *inetd*(8c) for more detail on the format of the configuration file and the operation of the Internet daemon.

### 5.7. Network data bases

Several data files are used by the network library routines and server programs. Most of these files are host independent and updated only rarely.

File	Manual reference	Use
<i>/etc/hosts</i>	<i>hosts</i> (5)	host names
<i>/etc/networks</i>	<i>networks</i> (5)	network names
<i>/etc/services</i>	<i>services</i> (5)	list of known services
<i>/etc/protocols</i>	<i>protocols</i> (5)	protocol names
<i>/etc/hosts.equiv</i>	<i>rshd</i> (8C)	list of "trusted" hosts
<i>/etc/rc.local</i>	<i>rc</i> (8)	command script for starting servers
<i>/etc/ftpusers</i>	<i>ftpd</i> (8C)	list of "unwelcome" ftp users
<i>/etc/hosts.lpd</i>	<i>lpd</i> (8C)	list of hosts allowed to access printers
<i>/etc/inetd.conf</i>	<i>inetd</i> (8)	list of servers started by <i>inetd</i>

The files distributed are set up for ARPANET or other Internet hosts. Local networks and hosts should be added to describe the local configuration; the Berkeley entries may serve as examples (see also the next section). Network numbers will have to be chosen for each Ethernet. For sites not connected to the Internet, these can be chosen more or less arbitrarily, otherwise the normal channels should be used for allocation of network numbers.

#### 5.7.1. Regenerating /etc/hosts and /etc/networks

When using the host address routines that use the Internet name server, the file */etc/hosts* is only used for setting interface addresses and at other times that the server is not running, and therefore it need only contain addresses for local hosts. There is no equivalent service for network names yet. The full host and network name data bases are normally derived from a file retrieved from the Internet Network Information Center at SRI. To do this you should use the program */etc/gettable* to retrieve the NIC host data base, and the program *htable*(8) to convert it to the format used by the libraries. You should change to the directory where you maintain your local additions to the host table and execute the following commands.

```
# /etc/gettable sri-nic.arpa
Connection to sri-nic.arpa opened.
Host table received.
Connection to sri-nic.arpa closed.
# /etc/htable hosts.txt
Warning, no localgateways file.
#
```

The *htable* program generates three files in the local directory: *hosts*, *networks* and *gateways*. If a file "localhosts" is present in the working directory its contents are first copied to the output file. Similarly, a "localnetworks" file may be prepended to the output created by *htable*, and "localgateways" will be prepended to *gateways*. It is usually wise to run *diff*(1) on the new host and network data bases before installing them in */etc*. If you are using the host table for host name and address mapping, you should run *mkhosts*(8) after installing */etc/hosts*. If you are using the name server for the host name and address mapping, you only need to install *networks* and a small copy of *hosts* describing your local machines. The full host table in this case might be placed somewhere else for reference by users. The *gateways* file may be installed in */etc/gateways* if you use *routed*(8c) for local routing and wish to have static external routes installed when *routed* is started. This procedure is essentially obsolete, however, except for individual hosts that are on the Arpanet or Milnet and do not forward packets from a local network. Other situations require the use of an EGP server.

If you are connected to the DARPA Internet, it is highly recommended that you use the name server for your host name and address mapping, as this provides access to a much larger set of hosts than are provided in the host table. Many large organization on the network, currently have only a small percentage of their hosts listed in the host table retrieved from NIC.

#### 5.7.2. /etc/hosts.equiv

The remote login and shell servers use an authentication scheme based on trusted hosts. The *hosts.equiv* file contains a list of hosts that are considered trusted and, under a single administrative control. When a user contacts a remote login or shell server requesting service, the client process passes the user's name and the official name of the host on which the client is located. In the simple case, if the host's name is located in *hosts.equiv* and the user has an account on the server's machine, then service is rendered (i.e. the user is allowed to log in, or the command is executed). Users may expand this "equivalence" of machines by installing a *.rhosts* file in their login directory. The root login is handled specially, bypassing the *hosts.equiv* file, and using only the *.rhosts* file.

Thus, to create a class of equivalent machines, the *hosts.equiv* file should contain the *official* names for those machines. If you are running the name server, you may omit the domain part of the host name for machines in your local domain. For example, several machines on our local network are considered trusted, so the *hosts.equiv* file is of the form:

ucbarpa  
calder  
dali  
ernie  
kim  
matisse  
monet  
ucbvax  
miro  
degas

### 5.7.3. /etc/rc.local

Most network servers are automatically started up at boot time by the command file /etc/rc (if they are installed in their presumed locations) or by the Internet daemon (see above). These include the following:

Program	Server	Started by
/etc/rshd	shell server	inetd
/etc/rexecd	exec server	inetd
/etc/rlogind	login server	inetd
/etc/telnetd	TELNET server	inetd
/etc/ftpd	FTP server	inetd
/etc/fingerd	Finger server	inetd
/etc/tftpd	TFTP server	inetd
/etc/rwhod	system status daemon	/etc/rc
/etc/syslogd	error logging server	/etc/rc
/usr/lib/sendmail	SMTP server	/etc/rc
/etc/routed	routing table management daemon	/etc/rc

Consult the manual pages and accompanying documentation (particularly for sendmail) for details about their operation.

To have other network servers started up as well, the appropriate line should be added to the Internet daemon's configuration file */etc/inetd.conf*, or commands of the following sort should be placed in the site dependent file */etc/rc.local*.

```
if [ -f /etc/routed ]; then
    /etc/routed & echo -n ' routed'      >/dev/console
fi
```

### 5.7.4. /etc/ftpusers

The FTP server included in the system provides support for an anonymous FTP account. Because of the inherent security problems with such a facility you should read this section carefully if you consider providing such a service.

An anonymous account is enabled by creating a user *ftp*. When a client uses the anonymous account a *chroot(2)* system call is performed by the server to restrict the client from moving outside that part of the file system where the user *ftp* home directory is located. Because a *chroot* call is used, certain programs and files used by the server process must be placed in the *ftp* home directory. Further, one must be sure that all directories and executable images are unwritable. The following directory setup is recommended.

```
# cd ~ftp
# chmod 555 .; chown ftp .; chgrp ftp .
# mkdir bin etc pub
# chown root bin etc
# chmod 555 bin etc
# chown ftp pub
# chmod 777 pub
# cd bin
# cp /bin/sh /bin/ls .
# chmod 111 sh ls
# cd ../etc
# cp /etc/passwd /etc/group .
# chmod 444 passwd group
```

When local users wish to place files in the anonymous area, they must be placed in a subdirectory. In the setup here, the directory *~ftp/pub* is used.

Another issue to consider is the copy of */etc/passwd* placed here. It may be copied by users who use the anonymous account. They may then try to break the passwords of users on your machine for further access. A good choice of users to include in this copy might be root, daemon, uucp, and the ftp user. All passwords here should probably be "\*\*\*".

Aside from the problems of directory modes and such, the ftp server may provide a loophole for interlopers if certain user accounts are allowed. The file */etc/ftpusers* is checked on each connection. If the requested user name is located in the file, the request for service is denied. This file normally has the following names on our systems.

```
uucp
root
```

Accounts with nonstandard shells should be listed in this file. Accounts without passwords need not be listed in this file, the ftp server will not service these users.



## 6. SYSTEM OPERATION

This section describes procedures used to operate a VAX UNIX system. Procedures described here are used periodically, to reboot the system, analyze error messages from devices, do disk back-ups, monitor system performance, recompile system software and control local changes.

### 6.1. Bootstrap and shutdown procedures

In a normal reboot, the system checks the disks and comes up multi-user without intervention at the console. Such a reboot can be stopped (after it prints the date) with a ^C (interrupt). This will leave the system in single-user mode, with only the console terminal active. It is also possible to allow the filesystem checks to complete and then to return to single-user mode by signaling *fsck* with a QUIT signal (^).

If booting from the console command level is needed, then the command

```
>>> B
```

will boot from the default device. On an 8650, 8600, 11/785, 11/780, or 11/730 the default device is determined by a "DEPOSIT" command stored on the console boot device in the file "DEFBOO.CMD" ("DEFBOO.COM" on an 8650 or 8600); on an 11/750 the default device is determined by the setting of a switch on the front panel.

You can boot a system up single user on an 8650, 8600, 785, 780, or 730 by doing

```
>>> B XXS
```

where *XX* is one of HP, HK, UP, RA, or RB for a 730. The corresponding command on an 11/750 is

```
>>> B/2
```

For second vendor storage modules on the UNIBUS or MASSBUS of an 11/750 you will need to have a boot prom. Most vendors will sell you such prompts for their controllers; contact your vendor if you don't have one.

Other possibilities are:

```
>>> B ANY
```

or, on a 750

```
>>> B/3
```

These commands boot and ask for the name of the system to be booted. They can be used after building a new test system to give the boot program the name of the test version of the system.

To bring the system up to a multi-user configuration from the single-user status after, e.g., a "B HPS" on an 8650, 8600, 11/785 or 11/780, "B RBS" on a 11/730, or a "B/2" on an 11/750 all you have to do is hit ^D on the console. The system will then execute */etc/rc*, a multi-user restart script (and */etc/rc.local*), and come up on the terminals listed as active in the file */etc/tty*s. See *init*(8) and *ttys*(5). Note, however, that this does not cause a file system check to be performed. Unless the system was taken down cleanly, you should run "fsck -p" or force a reboot with *reboot*(8) to have the disks checked.

To take the system down to a single user state you can use

```
# kill 1
```

or use the *shutdown*(8) command (which is much more polite, if there are other users logged in.) when you are up multi-user. Either command will kill all processes and give you a shell on the console, as if you had just booted. File systems remain mounted after the system is taken single-user. If you wish to come up multi-user again, you should do this by:

April 16, 1986

```
# cd /  
# /etc/umount -a  
# ^D
```

Each system shutdown, crash, processor halt and reboot is recorded in the file `/usr/adm/shutdownlog` with the cause.

## 6.2. Device errors and diagnostics

When serious errors occur on peripherals or in the system, the system prints a warning diagnostic on the console. These messages are collected by the system error logging process `syslogd(8)` and written into a system error log file `/usr/adm/messages`. Less serious errors are sent directly to `syslogd`, which may log them on the console. The error priorities that are logged and the locations to which they are logged are controlled by `/etc/syslog.conf`. See `syslogd(8)` for details.

Error messages printed by the devices in the system are described with the drivers for the devices in section 4 of the programmer's manual. If errors occur suggesting hardware problems, you should contact your hardware support group or field service. It is a good idea to examine the error log file regularly (e.g. with `"tail -r /usr/adm/messages"`).

## 6.3. File system checks, backups and disaster recovery

Periodically (say every week or so in the absence of any problems) and always (usually automatically) after a crash, all the file systems should be checked for consistency by `fsck(1)`. The procedures of `reboot(8)` should be used to get the system to a state where a file system check can be performed manually or automatically.

Dumping of the file systems should be done regularly, since once the system is going it is easy to become complacent. Complete and incremental dumps are easily done with `dump(8)`. You should arrange to do a towers-of-hanoi dump sequence; we tune ours so that almost all files are dumped on two tapes and kept for at least a week in most every case. We take full dumps every month (and keep these indefinitely). Operators can execute `"dump w"` at login that will tell them what needs to be dumped (based on the `/etc/fstab` information). Be sure to create a group operator in the file `/etc/group` so that `dump` can notify logged-in operators when it needs help.

More precisely, we have three sets of dump tapes: 10 daily tapes, 5 weekly sets of 2 tapes, and fresh sets of three tapes monthly. We do daily dumps circularly on the daily tapes with sequence `'3 2 5 4 7 6 9 8 9 9 9 ...'`. Each weekly is a level 1 and the daily dump sequence level restarts after each weekly dump. Full dumps are level 0 and the daily sequence restarts after each full dump also.

Thus a typical dump sequence would be:

April 16, 1986

tape name	level number	date	opr	size
FULL	0	Nov 24, 1979	jkf	137K
D1	3	Nov 28, 1979	jkf	29K
D2	2	Nov 29, 1979	rrh	34K
D3	5	Nov 30, 1979	rrh	19K
D4	4	Dec 1, 1979	rrh	22K
W1	1	Dec 2, 1979	etc	40K
D5	3	Dec 4, 1979	rrh	15K
D6	2	Dec 5, 1979	jkf	25K
D7	5	Dec 6, 1979	jkf	15K
D8	4	Dec 7, 1979	rrh	19K
W2	1	Dec 9, 1979	etc	118K
D9	3	Dec 11, 1979	rrh	15K
D10	2	Dec 12, 1979	rrh	26K
D1	5	Dec 15, 1979	rrh	14K
W3	1	Dec 17, 1979	etc	71K
D2	3	Dec 18, 1979	etc	13K
FULL	0	Dec 22, 1979	etc	135K

We do weekly dumps often enough that daily dumps always fit on one tape.

Dumping of files by name is best done by *tar*(1) but the amount of data that can be moved in this way is limited to a single tape. Finally if there are enough drives entire disks can be copied with *dd*(1) using the raw special files and an appropriate blocking factor; the number of sectors per track is usually a good value to use, consult */etc/disktab*.

It is desirable that full dumps of the root file system be made regularly. This is especially true when only one disk is available. Then, if the root file system is damaged by a hardware or software failure, you can rebuild a workable disk doing a restore in the same way that the initial root file system was created.

Exhaustion of user-file space is certain to occur now and then; disk quotas may be imposed, or if you prefer a less facist approach, try using the programs *du*(1), *df*(1), *quot*(8), combined with threatening messages of the day, and personal letters.

#### 6.4. Moving file system data

If you have the equipment, the best way to move a file system is to dump it to magtape using *dump*(8), use *newfs*(8) to create the new file system, and restore the tape, using *restore*(8). If for some reason you don't want to use magtape, *dump* accepts an argument telling where to put the dump; you might use another disk. Filesystems may also be moved by piping the output of *dump* to *restore*. The *restore* program uses an "in-place" algorithm that allows file system dumps to be restored without concern for the original size of the file system. Further, portions of a file system may be selectively restored using a method similar to the tape archive program.

If you have to merge a file system into another, existing one, the best bet is to use *tar*(1). If you must shrink a file system, the best bet is to dump the original and restore it onto the new file system. If you are playing with the root file system and only have one drive, the procedure is more complicated. If the only drive is a Winchester disk, this procedure may not be used without overwriting the existing root or another partition. What you do is the following:

1. GET A SECOND PACK!!!!
2. Dump the root file system to tape using *dump*(8).
3. Bring the system down and mount the new pack.
4. Load the distribution tape and install the new root file system as you did when first installing the system.

## 5. Boot normally using the newly created disk file system.

Note that if you change the disk partition tables or add new disk drivers they should also be added to the standalone system in */sys/stand* and the default disk partition tables in */etc/disktab* should be modified.

## 6.5. Monitoring System Performance

The *sysstat* program provided with the system is designed to be an aid to monitoring systemwide activity. The default "pigs" mode shows a dynamic "ps". By running in the "vmstat" mode when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, paging and swapping activity, device interrupts, and disk and cpu utilization. Ideally, there should be few blocked (b) jobs, there should be little paging or swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 20-30 tps in practice), and the user cpu utilization (us) should be high (above 50%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (b). If the virtual memory is active, then the paging demon will be running (sr will be non-zero). It is healthy for the paging demon to free pages when the virtual memory gets active; it is triggered by the amount of free memory dropping below a threshold and increases its pace as free memory goes to zero.

If you run in the "vmstat" mode when the system is busy, you can find imbalances by noting abnormal job distributions. If many processes are blocked (b), then the disk subsystem is overloaded or imbalanced. If you have several non-dma devices or open teletype lines that are "ringing", or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-70% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (cs), interrupt activity (in) and per-device interrupt counts, or system call activity (sy). Cumulatively on one of our large machines we average about 60-100 context switches and interrupts per second and about 70-120 system calls per second.

If the system is heavily loaded, or if you have little memory for your load (2M is little in most any case), then the system may be forced to swap. This is likely to be accompanied by a noticeable reduction in system performance and pregnant pauses when interactive jobs such as editors swap out. If you expect to be in a memory-poor environment for an extended period you might consider administratively limiting system load.

## 6.6. Recompiling and reinstalling system software

It is easy to regenerate the system, and it is a good idea to try rebuilding pieces of the system to build confidence in the procedures. The system consists of two major parts: the kernel itself (/sys) and the user programs (/usr/src and subdirectories). The major part of this is /usr/src.

The three major libraries are the C library in /usr/src/lib/libc and the FORTRAN libraries /usr/src/usr.lib/libl77 and /usr/src/usr.lib/libF77. In each case the library is remade by changing into the corresponding directory and doing

```
# make
```

and then installed by

```
# make install
```

Similar to the system,

```
# make clean
```

cleans up.

The source for all other libraries is kept in subdirectories of /usr/src/usr.lib; each has a makefile and can be recompiled by the above recipe.

If you look at /usr/src/Makefile, you will see that you can recompile the entire system source with one command. To recompile a specific program, find out where the source resides with the

*whereis*(1) command, then change to that directory and remake it with the makefile present in the directory. For instance, to recompile "date", all one has to do is

```
# whereis date
date: /usr/src/bin/date.c /bin/date /usr/man/man1/date.1
# cd /usr/src/bin
# make date
```

this will create an unstripped version of the binary of "date" in the current directory. To install the binary image, use the install command as in

```
# install -s date /bin/date
```

The *-s* option will insure the installed version of *date* has its symbol table stripped. The *install* command should be used instead of *mv* or *cp* as it understands how to install programs even when the program is currently in use.

If you wish to recompile and install all programs in a particular target area you can override the default target by doing:

```
# make
# make DESTDIR=pathname install
```

To regenerate all the system source you can do

```
# cd /usr/src
# make
```

If you modify the C library, say to change a system call, and want to rebuild and install everything from scratch you have to be a little careful. You must insure that the libraries are installed before the remainder of the source, otherwise the loaded images will not contain the new routine from the library. The following sequence will accomplish this.

```
# cd /usr/src
# make clean
# make build
# make installsrc
```

The first *make* removes any existing binaries in the source trees to insure that everything is reloaded. The next *make* compiles and installs the libraries and compilers, then compiles the remainder of the sources. The final line installs all of the commands not installed in the first phase. This will take about 18 hours on a reasonably configured 11/750.

## 6.7. Making local modifications

To keep track of changes to system source we migrate changed versions of commands in */usr/src/bin*, */usr/src/usr.bin*, and */usr/src/ucb* in through the directory */usr/src/new* and out of the original directory into */usr/src/old* for a time before removing them. (*/usr/new* is also used by default for the programs that constitute the contributed software portion of the distribution.) Locally written commands that aren't distributed are kept in */usr/src/local* and their binaries are kept in */usr/local*. This allows */usr/bin*, */usr/ucb*, and */bin* to correspond to the distribution tape (and to the manuals that people can buy). People wishing to use */usr/local* commands are made aware that they aren't in the base manual. As manual updates incorporate these commands they are moved to */usr/ucb*.

A directory, */usr/junk*, to throw garbage into, as well as binary directories, */usr/old* and */usr/new*, are useful. The *man* command supports manual directories such as */usr/man/man0* for old and */usr/man/man1* for local to make this or something similar practical.

### 6.8. Accounting

UNIX optionally records two kinds of accounting information: connect time accounting and process resource accounting. The connect time accounting information is stored in the file */usr/adm/wtmp*, which is summarized by the program *ac(8)*. The process time accounting information is stored in the file */usr/adm/acct* after it is enabled by *accton(8)*, and is analyzed and summarized by the program *sa(8)*.

If you need to recharge for computing time, you can develop procedures based on the information provided by these commands. A convenient way to do this is to give commands to the clock daemon */etc/cron* to be executed every day at a specified time. This is done by adding lines to */usr/adm/crontab*; see *cron(8)* for details.

### 6.9. Resource control

Resource control in the current version of UNIX is more elaborate than in most UNIX systems. The disk quota facilities developed at the University of Melbourne have been incorporated in the system and allow control over the number of files and amount of disk space each user may use on each file system. In addition, the resources consumed by any single process can be limited by the mechanisms of *setrlimit(2)*. As distributed, the latter mechanism is voluntary, though sites may choose to modify the login mechanism to impose limits not covered with disk quotas.

To use the disk quota facilities, the system must be configured with "options QUOTA". File systems may then be placed under the quota mechanism by creating a null file *quotas* at the root of the file system, running *quotacheck(8)*, and modifying */etc/fstab* to show that the file system is read-write with disk quotas (an "rq" type field). The *quotaon(8)* program may then be run to enable quotas.

Individual quotas are applied by using the quota editor *edquota(8)*. Users may view their quotas (but not those of other users) with the *quota(1)* program. The *repquota(8)* program may be used to summarize the quotas and current space usage on a particular file system or file systems.

Quotas are enforced with *soft* and *hard* limits. When a user first reaches a soft limit on a resource, a message is generated on his/her terminal. If the user fails to lower the resource usage below the soft limit the next time they log in to the system the *login* program will generate a warning about excessive usage. Should three login sessions go by with the soft limit breached the system then treats the soft limit as a *hard* limit and disallows any allocations until enough space is reclaimed to bring the user back below the soft limit. Hard limits are enforced strictly resulting in errors when a user tries to create or write a file. Each time a hard limit is exceeded the system will generate a message on the user's terminal.

Consult the auxiliary document, "Disc Quotas in a UNIX Environment" and the appropriate manual entries for more information.

### 6.10. Network troubleshooting

If you have anything more than a trivial network configuration, from time to time you are bound to run into problems. Before blaming the software, first check your network connections. On networks such as the Ethernet a loose cable tap or misplaced power cable can result in severely deteriorated service. The *netstat(1)* program may be of aid in tracking down hardware malfunctions. In particular, look at the *-i* and *-s* options in the manual page.

Should you believe a communication protocol problem exists, consult the protocol specifications and attempt to isolate the problem in a packet trace. The *SO\_DEBUG* option may be supplied before establishing a connection on a socket, in which case the system will trace all traffic and internal actions (such as timers expiring) in a circular trace buffer. This buffer may then be printed out with the *trpt(8C)* program. Most of the servers distributed with the system accept a *-d* option forcing all sockets to be created with debugging turned on. Consult the appropriate manual pages for more information.

### 6.11. Files that need periodic attention

We conclude the discussion of system operations by listing the files that require periodic attention or are system specific

<code>/etc/fstab</code>	how disk partitions are used
<code>/etc/disktab</code>	disk partition sizes
<code>/etc/printcap</code>	printer data base
<code>/etc/gettytab</code>	terminal type definitions
<code>/etc/remote</code>	names and phone numbers of remote machines for <i>tip(1)</i>
<code>/etc/group</code>	group memberships
<code>/etc/motd</code>	message of the day
<code>/etc/passwd</code>	password file; each account has a line
<code>/etc/rc.local</code>	local system restart script; runs reboot; starts daemons
<code>/etc/inetd.conf</code>	local internet servers
<code>/etc/hosts</code>	host name data base
<code>/etc/networks</code>	network name data base
<code>/etc/services</code>	network services data base
<code>/etc/hosts.equiv</code>	hosts under same administrative control
<code>/etc/syslog.conf</code>	error log configuration for <i>syslogd(8)</i>
<code>/etc/ttyd</code>	enables/disables ports
<code>/usr/lib/crontab</code>	commands that are run periodically
<code>/usr/lib/aliases</code>	mail forwarding and distribution groups
<code>/usr/adm/acct</code>	raw process account data
<code>/usr/adm/messages</code>	system error log
<code>/usr/adm/shutdownlog</code>	log of system reboots
<code>/usr/adm/wtmp</code>	login session accounting

April 16, 1986

## APPENDIX A - BOOTSTRAP DETAILS

This appendix contains pertinent files and numbers regarding the bootstrapping procedure for 4.3BSD. You should never have to look at this appendix. However, if there are problems in installing the distribution on your machine, the material contained here may prove useful.

### Contents of the distribution tape(s)

The distribution normally consists of three 1600bpi 2400' magnetic tapes or one 6250bpi 2400' magnetic tape. The layout of the 1600bpi tapes is listed below. The 6250bpi tape is in the same order, but is only on one tape. The first tape contains the following files on it. All tape files are blocked in 10 kilobytes records, except for the first file on the first tape that has 512 byte records.

Tape file	Records*	Contents
one	210	8 bootstrap monitor programs and a <i>tp(1)</i> file containing <i>boot</i> , <i>format</i> , and <i>copy</i>
two	205	"mini root" file system
three	430	<i>dump(8)</i> of distribution root file system
four	3000	<i>tar(1)</i> image of binaries and libraries in <i>/usr</i>

The second tape contains the following files:

Tape file	# Records	Contents
one	720	<i>tar(1)</i> image of <i>/sys</i> , including <i>GENERIC</i> system
two	2500	<i>tar(1)</i> image of <i>/usr/src</i>
three	580	<i>tar(1)</i> image of <i>/usr/lib/vfont</i>

The third tape contains the following files:

Tape file	# Records	Contents
one	3660	<i>tar(1)</i> image of user contributed software
two	250	<i>tar(1)</i> image of <i>/usr/ingres</i>

The distribution tape is made with the shell scripts located in the directory */sys/dist*. To build a distribution tape one must first create a mini root file system with the *buildmini* shell script.

\* The number of records in each tape file are approximate and do not correspond to the actual tape.



```
#!/bin/sh
#   @(#)buildmini 4.7 (Berkeley) 6/23/85
#
miniroot=hp0d
minitype=rm80
#
date
umount /dev/${miniroot}
newfs -s 4096 ${miniroot} ${minitype}
fsck /dev/r${miniroot}
mount /dev/${miniroot} /mnt
cd /mnt; sh /sys/dist/get
cd /sys/dist; sync
umount /dev/${miniroot}
fsck /dev/${miniroot}
date
```

The *buildmini* script uses the *get* script to build the file system.

```
#!/bin/sh
#
#   @(#)get 4.23 (Berkeley) 4/9/86
#
# Shell script to build a mini-root file system
# in preparation for building a distribution tape.
# The file system created here is image copied onto
# tape, then image copied onto disk as the "first"
# step in a cold boot of 4.2 systems.
#
DISTROOT=/nbsd
#
if [ `pwd` = '/' ]
then
    echo You just '(almost)' destroyed the root
    exit
fi
cp $DISTROOT/sys/GENERIC/vmunix .
rm -rf bin; mkdir bin
rm -rf etc; mkdir etc
rm -rf a; mkdir a
rm -rf tmp; mkdir tmp
rm -rf usr; mkdir usr usr/mdec
rm -rf sys; mkdir sys sys/floppy sys/cassette sys/console
cp $DISTROOT/etc/disktab etc
cp $DISTROOT/etc/newfs etc; strip etc/newfs
cp $DISTROOT/etc/mkfs etc; strip etc/mkfs
cp $DISTROOT/etc/restore etc; strip etc/restore
cp $DISTROOT/etc/init etc; strip etc/init
cp $DISTROOT/etc/mount etc; strip etc/mount
cp $DISTROOT/etc/mknod etc; strip etc/mknod
cp $DISTROOT/etc/fsck etc; strip etc/fsck
cp $DISTROOT/etc/umount etc; strip etc/umount
cp $DISTROOT/etc/arff etc; strip etc/arff
cp $DISTROOT/etc/fcopy etc; strip etc/fcopy
cp $DISTROOT/bin/mt bin; strip bin/mt
```

April 16, 1986

```
cp $DISTROOT/bin/ls bin; strip bin/ls
cp $DISTROOT/bin/sh bin; strip bin/sh
cp $DISTROOT/bin/mv bin; strip bin/mv
cp $DISTROOT/bin/sync bin; strip bin/sync
cp $DISTROOT/bin/cat bin; strip bin/cat
cp $DISTROOT/bin/mkdir bin; strip bin/mkdir
cp $DISTROOT/bin/stty bin; strip bin/stty; ln bin/stty bin/STTY
cp $DISTROOT/bin/echo bin; strip bin/echo
cp $DISTROOT/bin/rm bin; strip bin/rm
cp $DISTROOT/bin/cp bin; strip bin/cp
cp $DISTROOT/bin/expr bin; strip bin/expr
cp $DISTROOT/bin/[ bin; strip bin/[
cp $DISTROOT/bin/awk bin; strip bin/awk
cp $DISTROOT/bin/make bin; strip bin/make
cp $DISTROOT/usr/mdec/* usr/mdec
cp $DISTROOT/sys/floppy/[Ma-z0-9]* sys/floppy
cp $DISTROOT/sys/consolerl/[Ma-z0-9]* sys/consolerl
cp -r $DISTROOT/sys/cassette/[Ma-z0-9]* sys/cassette
cp $DISTROOT/sys/stand/boot boot
cp $DISTROOT/sys/stand/pcs750.bin pcs750.bin
cp $DISTROOT/.profile .profile
cat >etc/passwd <<EOF
root::0:10:::/bin/sh
EOF
cat >etc/group <<EOF
wheel:*:0:
staff:*:10:
EOF
cat >etc/fstab <<EOF
/dev/hp0a:/a:xx:1:1
/dev/up0a:/a:xx:1:1
/dev/hk0a:/a:xx:1:1
/dev/ra0a:/a:xx:1:1
/dev/rb0a:/a:xx:1:1
EOF
cat >xtr <<'EOF'
: ${disk?Usage: disk=xx0 type=tt tape=yy xtr'}
: ${type?Usage: disk=xx0 type=tt tape=yy xtr'}
: ${tape?Usage: disk=xx0 type=tt tape=yy xtr'}
echo 'Build root file system'
newfs ${disk}a ${type}
sync
echo 'Check the file system'
fsck /dev/r${disk}a
mount /dev/${disk}a /a
cd /a
echo 'Rewind tape'
mt -f /dev/${tape}0 rew
echo 'Restore the dump image of the root'
restore rsf 3 /dev/${tape}0
cd /
sync
umount /dev/${disk}a
sync
```

April 16, 1986

```

fsck /dev/r$(disk)a
echo 'Root filesystem extracted'
echo
echo 'If this is an 8650 or 8600, update the console r102'
echo 'If this is a 780 or 785, update the floppy'
echo 'If this is a 730, update the cassette'
EOF
chmod +x xtr
rm -rf dev; mkdir dev
cp $DISTROOT/sys/dist/MAKEDEV dev
chmod +x dev/MAKEDEV
cp /dev/null dev/MAKEDEV.local
cd dev
./MAKEDEV std hp0 hk0 up0 ra0 rb0
./MAKEDEV ts0; mv rmt12 ts0; rm *mt*;
./MAKEDEV tm0; mv rmt12 tm0; rm *mt*;
./MAKEDEV ht0; mv rmt12 ht0; rm *mt*;
./MAKEDEV ut0; mv rmt12 ut0; rm *mt*;
./MAKEDEV mt0; mv rmt4 xt0; rm *mt*; mv xt0 mt0
cd ..
sync

```

The mini root file system must have enough space to hold the files found on a floppy or cassette.

Once a mini root file system is constructed, the *maketape* script makes a distribution tape.

```

#!/bin/sh
#
#   @(#)maketape 4.27 (Berkeley) 10/17/85
#
#   maketape [ 6250 | 1600 [ tapename [ remotetapemachine ] ] ]
miniroot=hp0d
tape=/dev/rmt12
type=6250
if [ $# -gt 0 ]; then type=$1; fi
if [ $# -gt 1 ]; then tape=$2; fi
tartape=$tape
if [ $# -gt 2 ]; then remote=$3; tartape='-.'; fi
#
trap "rm -f /tmp/tape.$$; exit" 0 1 2 3 13 15
$remote mt -t ${tape} rew
date
umount /dev/hp2g
umount /dev/hp2a
mount -r /dev/hp2a /c/nbsd
mount -r /dev/hp2g /c/nbsd/usr
cd tp
tp cmf /tmp/tape.$$ boot copy format
cd /nbsd/sys/mdec
echo "Build 1st level boot block file"
cat tsboot htboot tmboot mtboot utboot noboot noboot /tmp/tape.$$ | \
    $remote dd of=${tape} obs=512 conv=sync
cd /nbsd
sync
echo "Add dump of mini-root file system"
eval dd if=/dev/r$(miniroot) count=205 bs=20b conv=sync ${remote+'|'} \

```

April 16, 1986

```

    ${remote+of=$tape"} ${remote+usr/local/20b ">" $tape'}
echo "Add full dump of real file system"
/etc/${remote+r}dump.0uf $remote${remote+:}${tape} /c/nbsd
echo "Add tar image of /usr"
cd /nbsd/usr; eval tar cf ${tartape} adm bin dict doc games \
    guest hosts include lib local man mdec msgs new \
    preserve pub spool tmp ucb \
    ${remote+} | $remote /usr/local/20b ">" $tape'
if [ $(type) != '6250' ]
then
    echo "Done, rewinding first tape"
    $remote mt -t ${tape} rew &
    echo "Mount second tape and hit return when ready"
    echo "(or type name of next tape drive)"
    read x
    if [ "$x" != "" ]
    then tape=$x
    fi
fi
echo "Add tar image of system sources"
cd /nbsd/sys; eval tar cf ${tartape} . \
    ${remote+} | $remote /usr/local/20b ">" $tape'
echo "Add user source code"
cd /nbsd/usr/src; eval tar cf ${tartape} Makefile bin etc games \
    include lib local old ucb undoc usr.bin usr.lib \
    ${remote+} | $remote /usr/local/20b ">" $tape'
echo "Add varian fonts"
cd /usr/lib/vfont; eval tar cf ${tartape} . \
    ${remote+} | $remote /usr/local/20b ">" $tape'
if [ $(type) != '6250' ]
then
    echo "Done, rewinding second tape"
    $remote mt -t ${tape} rew &
    echo "Mount third tape and hit return when ready"
    echo "(or type name of next tape drive)"
    read x
    if [ "$x" != "" ]
    then tape=$x
    fi
fi
echo "Add user contributed software"
cd /nbsd/usr/src/new; eval tar cf ${tartape} * \
    ${remote+} | $remote /usr/local/20b ">" $tape'
echo "Add ingres source"
cd /nbsd/usr/ingres; eval tar cf ${tartape} . \
    ${remote+} | $remote /usr/local/20b ">" $tape'
echo "Done, rewinding tape"
$remote mt -t ${tape} rew &

```

Summarizing then, to create a distribution tape you can use the above scripts and the following commands.

April 16, 1986

```
# buildmini
# maketape
...
(For 1600bpi tapes, the following will appear twice asking you to mount
fresh tapes)
Done, rewinding first tape
Mount second tape and hit return when ready
(remove the first tape and place a fresh one on the drive)
...
Done, rewinding second tape
```

#### Control status register addresses

The distribution uses many standalone device drivers that presume the location of a UNIBUS device's control status register (CSR). The following table summarizes these values.

Device name	Controller	CSR address (octal)
ra	DEC UDA50	0172150
rb	DEC 730 IDC	0175606
rk	DEC RK11	0177440
rl	DEC RL11	0174400
tm	EMULEX TC-11	0172520
ts	DEC TS11	0172520
up	EMULEX SC-21V	0176700
ut	SI 9700	0172440

All MASSBUS controllers are located at standard offsets from the base address of the MASSBUS adapter register bank.

#### Generic system control status register addresses

The *generic* version of the operating system supplied with the distribution contains the UNIBUS devices listed below. These devices will be recognized if the appropriate control status registers respond at any of the listed UNIBUS addresses.

Device name	Controller	CSR addresses (octal)
hk	DEC RK11	0177440
tm	EMULEX TC-11	0172520
tmscp	DEC TU81, TMSCP	0174500
ts	DEC TS11	0172520
ut	SI 9700	0172440
up	EMULEX SC-21V	0176700, 0174400, 0176300
ra	DEC UDA-50	0172150, 0172550, 0177550
rb	DEC 730 IDC	0175606
rl	DEC RL11	0174400
dm	DM11 equivalent	0170500
dh	DH11 equivalent	0160020, 0160040
dhu	DEC DHU11	0160440, 0160500
dz	DEC DZ11	0160100, 0160110, ... 0160170
dmf	DEC DMF32	0160340
dmz	DEC DMZ32	0160540
lp	DEC LP11	0177514
en	Xerox 3MB ethernet	0161000
ec	3Com ethernet	0164330
ex	Excelan ethernet	0164344
il	Interlan ethernet	0164000
de	DEC DEUNA	0174510

If devices other than the above are located at any of the addresses listed, the system may not bootstrap properly.

## APPENDIX B – LOADING THE TAPE MONITOR

This section describes how the bootstrap monitor located on the first tape of the distribution tape set may be loaded. This should not be necessary, but has been included as a fallback measure if it is not possible to read the distributed console medium. **WARNING:** the bootstraps supplied below may not work, in certain instances on an 11/730 because they use a buffered data path for transferring data from tape to memory; consult our group if you are unable to load the monitor on an 11/730. All of the addresses given below refer to the first SBIA on the 8650 and 8600.

To load the tape bootstrap monitor, first mount the magnetic tape on drive 0 at load point, making sure that the write ring is not inserted. Temporarily set the reboot switch on an 11/785, 11/780, or 11/730 to off; on an 8650, 8600, or 11/750 set the power-on action to halt. (In normal operation an 11/785, 11/780, or 11/730 will have the reboot switch on, and an 8650, 8600, or 11/750 will have the power-on action set to boot/restart.)

If you have an 8650, 8600, 11/785 or 11/780 give the commands:

```
>>> HALT
>>> UNJAM
```

Then, on any machine, give the init command:

```
>>> I
```

and then key in at location 200 and execute either the TS, HT, TM, or MT bootstrap that follows, as appropriate. **NOTE:** All of the addresses given in this section refer to the first SBIA on the 8650 and 8600. The machine's printouts are shown in boldface, explanatory comments are within ( ). (You can use 'delete' to erase a character and 'control U' to kill the whole line.)

### TS bootstrap

```
>>> D/P 200 3AEFD0
>>> D + D05A0000
>>> D + 3BEF
>>> D + 800CA00
>>> D + 32EFC1
>>> D + CA010000
>>> D + EFC10804
>>> D + 24
>>> D + 15508F
>>> D + ABB45B00
>>> D + 2AB9502
>>> D + 8FB0FB18
>>> D + 956B024C
>>> D + FB1802AB
>>> D + 25C8FB0
>>> D + 6B
```

(The next two deposits set up the addresses of the UNIBUS)  
(adapter and its memory; the 20006000 here is the address of)  
(uba0 and the 2013E000 the address of the I/O page, umem0)  
(on an 8650, 8600, 11/785 or 11/780)

```
>>> D + 20006000      (8650/8600/785/780 uba0)
                      (8650/8600/785/780 uba1: 20008000, uba2 2000A000)
                      (750 uba0: F30000, uba1: F32000; 730 uba: F26000)
>>> D + 2013E000      (8650/8600/785/780 umem0)
                      (8650/8600/785/780 umem1: 2017E000, umem2: 201BE000)
```

April 16, 1986

(750 umem0: FFE000, umem1: FBE000; 730 umem: FFE000)

```
>>> D + 80000000
>>> D + 254C004
>>> D + 80000
>>> D + 264
>>> D + E
>>> D + C001
>>> D + 2000000
>>> S 200
```

#### HT bootstrap

```
>>> D/P 200 3EEFD0
>>> D + C55A0000
>>> D + 3BEF
>>> D + 808F00
>>> D + C15B0000
>>> D + C05B5A5B
>>> D + 4008F
>>> D + D05B00
>>> D + 9D004AA
>>> D + C08F326B
>>> D + D424AB14
>>> D + 8FD00CAA
>>> D + 80000000
>>> D + 320800CA
>>> D + AAFE008F
>>> D + 6B39D010
>>> D + 0
    (The next two deposits set up the addresses of the MASSBUS)
    (adapter and the drive number for the tape formatter)
    (the 20010000 here is the address of mba0 on an 8650, 8600, 11/785,)
    (or 11/780 and the 0 indicates the formatter is drive 0 on mba0)
>>> D + 20010000      (8650/8600/785/780 mba0)
    (8650/8600/785/780 mba1: 20012000; 750 mba0: F28000, mba1: F2A000)
>>> D + 0              (Formatter unit number in range 0-7)
>>> S 200
>>> S 200
```

#### TM bootstrap

```
>>> D/P 200 2AEFD0
>>> D + D0510000
>>> D + 2000008F
>>> D + 800C180
>>> D + 804C1D4
>>> D + 1AEFD0
>>> D + C8520000
>>> D + F5508F
>>> D + 8FAE5200
>>> D + 4A20200
>>> D + B006A2B4
>>> D + 2A203
    (The following two numbers are uba0 and umem0 on a 8650/8600/785/780)
```



(See TS above for values for other CPU's and UBA's)  
>>> D + 20006000 (8650/8600/785/780 uba0)  
>>> D + 2013E000 (8650/8600/785/780 umem0)  
>>> S 200  
>>> S 200  
>>> S 200

#### MT bootstrap

>>> D/P 200 46EFD0  
>>> D + C55A0000  
>>> D + 43EF  
>>> D + 808F00  
>>> D + C15B0000  
>>> D + C05B5A5B  
>>> D + 4008F  
>>> D + 19A5B00  
>>> D + 49A04AA  
>>> D + AAD408AB  
>>> D + 8FD00C  
>>> D + CA800000  
>>> D + 8F320800  
>>> D + 10AAFE00  
>>> D + 2008F3C  
>>> D + ABD014AB  
>>> D + FE15044  
>>> D + 399AF850  
>>> D + 6B  
(The next two deposits set up the addresses of the MASSBUS)  
(adapter and the drive number for the tape formatter)  
(the 20012000 here is the address of mbal on an 8650, 8600, 11/785)  
(or 11/780 and the 0 indicates the formatter is drive 0 on mbal)  
>>> D + 20012000  
>>> D + 0  
>>> S 200  
>>> S 200  
>>> S 200  
>>> S 200

(no functioning toggle-in code exists for the UT device)

If the tape doesn't move the first time you start the bootstrap program with "S 200" you probably have entered the program incorrectly (but also check that the tape is online). Start over and check your typing. For the HT, MT, and TM bootstraps you will not be able to see the tape motion as you advance through the first few blocks as the tape motion is all within the vacuum columns.

Next, deposit in R10 the address of the tape MBA/UBA and in R11 the address of the device registers or unit number from one of:

April 16, 1986

```
>>> D/G A 20006000 (for tapes on 8650/8600/785/780 uba0)
>>> D/G A 20008000 (for tapes on 8650/8600/785/780 uba1)
>>> D/G A 20010000 (for tapes on 8650/8600/785/780 mba0)
>>> D/G A 20012000 (for tapes on 8650/8600/785/780 mba1)
>>> D/G A F30000 (for tapes on 750 uba0)
>>> D/G A F32000 (for tapes on 750 uba1)
>>> D/G A F28000 (for tapes on 750 mba0)
>>> D/G A F2A000 (for tapes on 750 mba1)
>>> D/G A F26000 (for tapes on 730 uba0)
```

and for register 11:

```
>>> D/G B 0 (for TM03/TM78 formatters at mba? drive 0)
>>> D/G B 1 (for TM03/TM78 formatters at mba? drive 1)
>>> D/G B 2013F550 (for TM11/TS11/TU80 tapes on 8650/8600/785/780 uba0)
>>> D/G B FFF550 (for TM11/TS11/TU80 tapes on 750 or 730 uba0)
```

Then start the bootstrap program with

```
>>> S 0
```

The console should type

=

You are now talking to the tape bootstrap monitor. At any point in the following procedure you can return to this section, reload the tape bootstrap, and restart the procedure. The console monitor is identical to that loaded from a TU58 console cassette, follow the instructions in section 2 as they apply to this device. The only exception is that when using programs loaded from the tape bootstrap monitor, programs will always return to the monitor (the "=" prompt). This saves your having to type in the above toggle-in code for each program to be loaded.

April 16, 1986

## APPENDIX C - INSTALLATION TROUBLESHOOTING

This appendix lists and explains certain problems that might be encountered while trying to install the 4.3BSD distribution. The information provided here is limited to the early steps in the installation process; i.e. up to the point where the root file system is installed. If you have a problem installing the release consult this section before contacting our group.

### Using the distribution console medium.

This section describes problems that may occur when using the programs provided on the distributed console medium: TU58 cassette or RX01 floppy disk.

*program can not be loaded.*

Check to make sure the correct floppy or cassette is being used. If using a floppy, be sure it is not in upside down. If using a cassette on an 11/730, be certain drive 0 is being used. If a hard I/O error occurred while reading a floppy, try resetting the console LSI-11 by powering it on and off. If you can not boot the cassette's bootstrap monitor, verify that the standard DEC console cassette can be read; if it can not, your cassette drive is probably broken.

*program halts without warning.*

Check to make sure you have specified the correct disk to format; consult sections 1.3 and 1.4 for a discussion of the VAX and UNIX device naming conventions. On 11/750's, specifying a non-existent MASSBUS device will cause the program to halt as it receives an interrupt (standalone programs operate by polling devices).

If using a floppy, try reading the floppy under your current system. If this works, copy the floppy to a new one and begin again. If using a cassette on an 11/730, do likewise.

*format prints "Known devices are ...".*

You have requested *format* to work on a device for which it has no driver, or that does not exist; only the listed devices are supported.

*format, boot, or copy prints "unknown drive type".*

A MASSBUS disk was specified, but the associated MASSBUS drive type register indicates a drive of unknown type. This probably means you typed something wrong or your hardware is incorrectly configured.

*format, boot, or copy prints "unknown device".*

The device specified is probably not one of those supported by the distribution; consult section 1.1. If the device is listed in section 1.1, the drive may be dual-ported, or for some other reason the driver was unable to decipher its characteristics. If this is a MASSBUS drive, try powering the MASSBUS adapter and/or controller on and off to clear the drive type register.

*copy does not copy 205 records*

If a tape read error occurred, clean your tape drive heads. If a disk write error occurred, the disk formatting may have failed. If the disk pack is removable, try another one. If you are currently running UNIX, you can reboot your old system and use *dd* to copy the mini-root file system into a disk partition (assuming the destination is not in use by the running system).

*boot prints "not a directory"*

The *boot* program was unable to find the requested program because it encountered something other than a directory while searching the file system. This usually suggests that no file system is present on the disk partition supplied, or the file system has been corrupted. First check to make sure you typed the correct line to boot. If this is the case and you are booting from the mini-root file system, the mini-root was probably not copied correctly off the tape (perhaps it was not placed in the correct disk partition). Try reinstalling the mini-root file system or, if trying to boot the true root file system, try booting from the mini-root file system and run *fsck* on the restored root file system to insure its

integrity. Finally, as a last resort, copy the *boot* program from the mini-root file system to the newly installed root file system.

*boot prints "bad format"*

The program you requested *boot* to load did not have a 407, 410, or 413 magic number in its header. This should never happen on a distribution system. If you were trying to boot off the root file system, reboot the system on the mini-root file system and look at the program on the root file system. Try copying the copy of *vmunix* on the mini-root to the root file system also.

*boot prints "read short"*

The file header for the program contained a size larger than the actual size of the file located on disk. This is probably the result of file system corruption (or a disk I/O error). Try booting again or creating a new copy of the program to be loaded (see above).

#### Booting the generic system

This section contains common problems encountered when booting the generic version of the system.

*system panics with "panic: iinit"*

This occurred because the system was unable to mount the root file system. The root file system supplied at the "root device?" prompt was probably incorrect. Remember that when running on the mini-root file system, this question must be answered with something of the form "hp0\*". If the answer had been "hp0", the system would have used the "a" partition on unit 0 of the "hp" drive, where presumably no file system exists.

Alternatively, the file system on which you were trying to run is corrupted. Try reinstalling the appropriate file system.

*system selects incorrect root device*

That is, you try to boot the system single user with "B/2" or "B xxS" but do not get the root file system in the expected location. This is most likely caused by your having many disks available more suited to be a root file system than the one you wanted. For example, if you have a "up" disk and an "hk" disk and install the system on the "hk", then try to boot the system to single-user mode, the heuristic used by the generic system to select the root file system will choose the "up" disk. The following list gives, in descending order, those disks thought most suitable to be a root file system: "hp", "up", "ra", "rb", "rl", "hk" (the position of "rl" is subject to argument). To get the root device you want you must boot using "B/3" or "B ANY", then supply the root device at the prompt.

*system crashes during autoconfiguration*

This is almost always caused by an unsupported UNIBUS device being present at a location where a supported device was expected. You must disable the device in some way, either by pulling it off the bus, or by moving the location of the console status register (consult Appendix A for a complete list of UNIBUS csr's used in the generic system).

*system does not find device(s)*

The UNIBUS device is not at a standard location. Consult the list of control status register addresses in Appendix A, or wait to configure a system to your hardware.

Alternatively, certain devices are difficult to locate during autoconfiguration. A classic example is the TS11 tape drive that does not autoconfigure properly if it is rewinding when the system is rebooted. Tape drives should configure properly if they are off-line, or are not performing a tape movement. Disks that are dual-ported should autoconfigure properly if the drive is not being simultaneously accessed through the alternate port.

#### Building console cassettes

This sections describes common problems encountered while constructing a console bootstrap cassette.

*system crashes*

You are trying to build a cassette for an 11/750. On an 11/750 the system is booted by using a bootstrap prom and sector 0 of the root file system. Refer to section 2.1.5 or *tu(4)* for the appropriate reprimand.

*system hangs*

You are using an MRSP prom on an 11/750 and think you can ignore the instructions in this document. The problem here is that the generic system only supports the MRSP prom on an 11/730. Using it on an 11/750 requires a special system configuration; consult *tu(4)* for more information.

April 16, 1986



## Building Berkeley UNIX<sup>†</sup> Kernels with Config

*Samuel J. Leffler and Michael J. Karels*

Computer Systems Research Group  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California 94720

### ABSTRACT

This document describes the use of *config*(8) to configure and create bootable 4.3BSD system images. It discusses the structure of system configuration files and how to configure systems with non-standard hardware configurations. Sections describing the preferred way to add new code to the system and how the system's autoconfiguration process operates are included. An appendix contains a summary of the rules used by the system in calculating the size of system data structures, and also indicates some of the standard system size limitations (and how to change them). Other configuration options are also listed.

Revised June 3, 1986

### 1. INTRODUCTION

*Config* is a tool used in building 4.3BSD system images (the UNIX kernel). It takes a file describing a system's tunable parameters and hardware support, and generates a collection of files which are then used to build a copy of UNIX appropriate to that configuration. *Config* simplifies system maintenance by isolating system dependencies in a single, easy to understand, file.

This document describes the content and format of system configuration files and the rules which must be followed when creating these files. Example configuration files are constructed and discussed.

Later sections suggest guidelines to be used in modifying system source and explain some of the inner workings of the autoconfiguration process. Appendix D summarizes the rules used in calculating the most important system data structures and indicates some inherent system data structure size limitations (and how to go about modifying them).

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

## 2. CONFIGURATION FILE CONTENTS

A system configuration must include at least the following pieces of information:

- machine type
- cpu type
- system identification
- timezone
- maximum number of users
- location of the root file system
- available hardware

*Config* allows multiple system images to be generated from a single configuration description. Each system image is configured for identical hardware, but may have different locations for the root file system and, possibly, other system devices.

### 2.1. Machine type

The *machine type* indicates if the system is going to operate on a DEC VAX-11† computer, or some other machine on which 4.3BSD operates. The machine type is used to locate certain data files which are machine specific, and also to select rules used in constructing the resultant configuration files.

### 2.2. Cpu type

The *cpu type* indicates which, of possibly many, cpu's the system is to operate on. For example, if the system is being configured for a VAX-11, it could be running on a VAX 8600, VAX-11/780, VAX-11/750, VAX-11/730 or MicroVAX II. (Other VAX cpu types, including the 8650, 785 and 725, are configured using the cpu designation for compatible machines introduced earlier.) Specifying more than one cpu type implies that the system should be configured to run on any of the cpu's specified. For some types of machines this is not possible and *config* will print a diagnostic indicating such.

### 2.3. System identification

The *system identification* is a moniker attached to the system, and often the machine on which the system is to run. For example, at Berkeley we have machines named Ernie (Co-VAX), Kim (No-VAX), and so on. The system identifier selected is used to create a global C “#define” which may be used to isolate system dependent pieces of code in the kernel. For example, Ernie's Varian driver used to be special cased because its interrupt vectors were wired together. The code in the driver which understood how to handle this non-standard hardware configuration was conditionally compiled in only if the system was for Ernie.

The system identifier “GENERIC” is given to a system which will run on any cpu of a particular machine type; it should not otherwise be used for a system identifier.

### 2.4. Timezone

The timezone in which the system is to run is used to define the information returned by the *gettimeofday(2)* system call. This value is specified as the number of hours east or west of GMT. Negative numbers indicate a value east of GMT. The timezone specification may also indicate the type of daylight savings time rules to be applied.

† DEC, VAX, UNIBUS, MASSBUS and MicroVAX are trademarks of Digital Equipment Corporation.





## 2.5. Maximum number of users

The system allocates many system data structures at boot time based on the maximum number of users the system will support. This number is normally between 8 and 40, depending on the hardware and expected job mix. The rules used to calculate system data structures are discussed in Appendix D.

## 2.6. Root file system location

When the system boots it must know the location of the root of the file system tree. This location and the part(s) of the disk(s) to be used for paging and swapping must be specified in order to create a complete configuration description. *Config* uses many rules to calculate default locations for these items; these are described in Appendix B.

When a generic system is configured, the root file system is left undefined until the system is booted. In this case, the root file system need not be specified, only that the system is a generic system.

## 2.7. Hardware devices

When the system boots it goes through an *autoconfiguration* phase. During this period, the system searches for all those hardware devices which the system builder has indicated might be present. This probing sequence requires certain pieces of information such as register addresses, bus interconnects, etc. A system's hardware may be configured in a very flexible manner or be specified without any flexibility whatsoever. Most people do not configure hardware devices into the system unless they are currently present on the machine, expect them to be present in the near future, or are simply guarding against a hardware failure somewhere else at the site (it is often wise to configure in extra disks in case an emergency requires moving one off a machine which has hardware problems).

The specification of hardware devices usually occupies the majority of the configuration file. As such, a large portion of this document will be spent understanding it. Section 6.3 contains a description of the autoconfiguration process, as it applies to those planning to write, or modify existing, device drivers.

## 2.8. Pseudo devices

Several system facilities are configured in a manner like that used for hardware devices although they are not associated with specific hardware. These system options are configured as *pseudo-devices*. Some pseudo devices allow an optional parameter that sets the limit on the number of instances of the device that are active simultaneously.

## 2.9. System options

Other than the mandatory pieces of information described above, it is also possible to include various optional system facilities or to modify system behavior and/or limits. For example, 4.3BSD can be configured to support binary compatibility for programs built under 4.1BSD. Also, optional support is provided for disk quotas and tracing the performance of the virtual memory subsystem. Any optional facilities to be configured into the system are specified in the configuration file. The resultant files generated by *config* will automatically include the necessary pieces of the system.

### 3. SYSTEM BUILDING PROCESS

In this section we consider the steps necessary to build a bootable system image. We assume the system source is located in the "/sys" directory and that, initially, the system is being configured from source code.

Under normal circumstances there are 5 steps in building a system.

- 1) Create a configuration file for the system.
- 2) Make a directory for the system to be constructed in.
- 3) Run *config* on the configuration file to generate the files required to compile and load the system image.
- 4) Construct the source code interdependency rules for the configured system with *makedepend* using *make(1)*.
- 5) Compile and load the system with *make*.

Steps 1 and 2 are usually done only once. When a system configuration changes it usually suffices to just run *config* on the modified configuration file, rebuild the source code dependencies, and remake the system. Sometimes, however, configuration dependencies may not be noticed in which case it is necessary to clean out the relocatable object files saved in the system's directory; this will be discussed later.

#### 3.1. Creating a configuration file

Configuration files normally reside in the directory "/sys/conf". A configuration file is most easily constructed by copying an existing configuration file and modifying it. The 4.3BSD distribution contains a number of configuration files for machines at Berkeley; one may be suitable or, in worst case, a copy of the generic configuration file may be edited.

The configuration file must have the same name as the directory in which the configured system is to be built. Further, *config* assumes this directory is located in the parent directory of the directory in which it is run. For example, the generic system has a configuration file "/sys/conf/GENERIC" and an accompanying directory named "/sys/GENERIC". Although it is not required that the system sources and configuration files reside in "/sys," the configuration and compilation procedure depends on the relative locations of directories within that hierarchy, as most of the system code and the files created by *config* use pathnames of the form "...". If the system files are not located in "/sys," it is desirable to make a symbolic link there for use in installation of other parts of the system that share files with the kernel.

When building the configuration file, be sure to include the items described in section 2. In particular, the machine type, cpu type, timezone, system identifier, maximum users, and root device must be specified. The specification of the hardware present may take a bit of work; particularly if your hardware is configured at non-standard places (e.g. device registers located at funny places or devices not supported by the system). Section 4 of this document gives a detailed description of the configuration file syntax, section 5 explains some sample configuration files, and section 6 discusses how to add new devices to the system. If the devices to be configured are not already described in one of the existing configuration files you should check the manual pages in section 4 of the UNIX Programmers Manual. For each supported device, the manual page synopsis entry gives a sample configuration line.

Once the configuration file is complete, run it through *config* and look for any errors. Never try and use a system which *config* has complained about; the results are unpredictable. For the most part, *config*'s error diagnostics are self explanatory. It may be the case that the line numbers given with the error messages are off by one.

A successful run of *config* on your configuration file will generate a number of files in the configuration directory. These files are:

- A file to be used by *make(1)* in compiling and loading the system, *Makefile*.

- One file for each possible system image for this machine, *swapxxx.c*, where *xxx* is the name of the system image, which describes where swapping, the root file system, and other miscellaneous system devices are located.
- A collection of header files, one per possible device the system supports, which define the hardware configured.
- A file containing the I/O configuration tables used by the system during its *autoconfiguration* phase, *ioconf.c*.
- An assembly language file of interrupt vectors which connect interrupts from the machine's external buses to the main system path for handling interrupts, and a file that contains counters and names for the interrupt vectors.

Unless you have reason to doubt *config*, or are curious how the system's autoconfiguration scheme works, you should never have to look at any of these files.

### 3.2. Constructing source code dependencies

When *config* is done generating the files needed to compile and link your system it will terminate with a message of the form "Don't forget to run make depend". This is a reminder that you should change over to the configuration directory for the system just configured and type "make depend" to build the rules used by *make* to recognize interdependencies in the system source code. This will insure that any changes to a piece of the system source code will result in the proper modules being recompiled the next time *make* is run.

This step is particularly important if your site makes changes to the system include files. The rules generated specify which source code files are dependent on which include files. Without these rules, *make* will not recognize when it must rebuild modules due to the modification of a system header file. The dependency rules are generated by a pass of the C preprocessor and reflect the global system options. This step must be repeated when the configuration file is changed and *config* is used to regenerate the system makefile.

### 3.3. Building the system

The makefile constructed by *config* should allow a new system to be rebuilt by simply typing "make image-name". For example, if you have named your bootable system image "vmunix", then "make vmunix" will generate a bootable image named "vmunix". Alternate system image names are used when the root file system location and/or swapping configuration is done in more than one way. The makefile which *config* creates has entry points for each system image defined in the configuration file. Thus, if you have configured "vmunix" to be a system with the root file system on an "hp" device and "hkvmmunix" to be a system with the root file system on an "hk" device, then "make vmunix hkvmmunix" will generate binary images for each. As the system will generally use the disk from which it is loaded as the root filesystem, separate system images are only required to support different swap configurations.

Note that the name of a bootable image is different from the system identifier. All bootable images are configured for the same system; only the information about the root file system and paging devices differ. (This is described in more detail in section 4.)

The last step in the system building process is to rearrange certain commonly used symbols in the symbol table of the system image; the makefile generated by *config* does this automatically for you. This is advantageous for programs such as *netstat(1)* and *vmstat(1)*, which run much faster when the symbols they need are located at the front of the symbol table. Remember also that many programs expect the currently executing system to be named "vmunix". If you install a new system and name it something other than "vmunix", many programs are likely to give strange results.

### 3.4. Sharing object modules

If you have many systems which are all built on a single machine there are at least two approaches to saving time in building system images. The best way is to have a single system image

which is run on all machines. This is attractive since it minimizes disk space used and time required to rebuild systems after making changes. However, it is often the case that one or more systems will require a separately configured system image. This may be due to limited memory (building a system with many unused device drivers can be expensive), or to configuration requirements (one machine may be a development machine where disk quotas are not needed, while another is a production machine where they are), etc. In these cases it is possible for common systems to share relocatable object modules which are not configuration dependent; most of the modules in the directory `"/sys/sys"` are of this sort.

To share object modules, a generic system should be built. Then, for each system configure the system as before, but before recompiling and linking the system, type `"make links"` in the system compilation directory. This will cause the system to be searched for source modules which are safe to share between systems and generate symbolic links in the current directory to the appropriate object modules in the directory `"../GENERIC"`. A shell script, `"makelinks"` is generated with this request and may be checked for correctness. The file `"/sys/conf/defines"` contains a list of symbols which we believe are safe to ignore when checking the source code for modules which may be shared. Note that this list includes the definitions used to conditionally compile in the virtual memory tracing facilities, and the trace point support used only rarely (even at Berkeley). It may be necessary to modify this file to reflect local needs. Note further that interdependencies which are not directly visible in the source code are not caught. This means that if you place per-system dependencies in an include file, they will not be recognized and the shared code may be selected in an unexpected fashion.

### 3.5. Building profiled systems

It is simple to configure a system which will automatically collect profiling information as it operates. The profiling data may be collected with *kgmon*(8) and processed with *gprof*(1) to obtain information regarding the system's operation. Profiled systems maintain histograms of the program counter as well as the number of invocations of each routine. The *gprof* command will also generate a dynamic call graph of the executing system and propagate time spent in each routine along the arcs of the call graph (consult the *gprof* documentation for elaboration). The program counter sampling can be driven by the system clock, or if you have an alternate real time clock, this can be used. The latter is highly recommended, as use of the system clock will result in statistical anomalies, and time spent in the clock routine will not be accurately attributed.

To configure a profiled system, the `-p` option should be supplied to *config*. A profiled system is about 5-10% larger in its text space due to the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer which is 1.2 times the size of the text space. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code.

Note that systems configured for profiling should not be shared as described above unless all the other shared systems are also to be profiled.

## 4. CONFIGURATION FILE SYNTAX

In this section we consider the specific rules used in writing a configuration file. A complete grammar for the input language can be found in Appendix A and may be of use if you should have problems with syntax errors.

A configuration file is broken up into three logical pieces:

- configuration parameters global to all system images specified in the configuration file,
- parameters specific to each system image to be generated, and
- device specifications.

### 4.1. Global configuration parameters

The global configuration parameters are the type of machine, cpu types, options, timezone, system identifier, and maximum users. Each is specified with a separate line in the configuration file.

**machine type**

The system is to run on the machine type specified. No more than one machine type can appear in the configuration file. Legal values are *vax* and *sun*.

**cpu "type"**

This system is to run on the cpu type specified. More than one cpu type specification can appear in a configuration file. Legal types for a *vax* machine are VAX8600, VAX780, VAX750, VAX730 and VAX630 (MicroVAX II). The 8650 is listed as an 8600, the 785 as a 780, and a 725 as a 730.

**options optionlist**

Compile the listed optional code into the system. Options in this list are separated by commas. Possible options are listed at the top of the generic makefile. A line of the form "options FUNNY,HAHA" generates global "#define"s -DFUNNY -DHAHA in the resultant makefile. An option may be given a value by following its name with "=", then the value enclosed in (double) quotes. The following are major options currently in use: COMPAT (include code for compatibility with 4.1BSD binaries), INET (Internet communication protocols), NS (Xerox NS communication protocols), and QUOTA (enable disk quotas). Other kernel options controlling system sizes and limits are listed in Appendix D; options for the network are found in Appendix E. There are additional options which are associated with certain peripheral devices; those are listed in the Synopsis section of the manual page for the device.

**makeoptions optionlist**

Options that are used within the system makefile and evaluated by *make* are listed as *makeoptions*. Options are listed with their values with the form "makeoptions name=value,name2=value2." The values must be enclosed in double quotes if they include numerals or begin with a dash.

**timezone number [ dst [ number ] ]**

Specifies the timezone used by the system. This is measured in the number of hours your timezone is west of GMT. EST is 5 hours west of GMT, PST is 8. Negative numbers indicate hours east of GMT. If you specify *dst*, the system will operate under daylight savings time. An optional integer or floating point number may be included to specify a particular daylight saving time correction algorithm; the default value is 1, indicating the United States. Other values are: 2 (Australian style), 3 (Western European), 4 (Middle European), and 5 (Eastern European). See *gettimeofday(2)* and *ctime(3)* for more information.

**ident name**

This system is to be known as *name*. This is usually a cute name like ERNIE (short for Ernie Co-Vax) or VAXWELL (for Vaxwell Smart). This value is defined for use in conditional compilation, and is also used to locate an optional list of source files specific to this system.

**maxusers number**

The maximum expected number of simultaneously active user on this system is *number*. This number is used to size several system data structures.

**4.2. System image parameters**

Multiple bootable images may be specified in a single configuration file. The systems will have the same global configuration parameters and devices, but the location of the root file system and other system specific devices may be different. A system image is specified with a "config" line:

**config sysname config-clauses**

The *sysname* field is the name given to the loaded system image; almost everyone names their standard system image "vmunix". The configuration clauses are one or more specifications indicating where the root file system is located and the number and location of paging devices. The device used by the system to process argument lists during *execve(2)* calls may also be specified, though in practice this is almost always selected by *config* using one of its rules for selecting default locations for system devices.



A configuration clause is one of the following

```
root [ on ] root-device
swap [ on ] swap-device [ and swap-device ] ...
dumps [ on ] dump-device
args [ on ] arg-device
```

(the “on” is optional.) Multiple configuration clauses are separated by white space; *config* allows specifications to be continued across multiple lines by beginning the continuation line with a tab character. The “root” clause specifies where the root file system is located, the “swap” clause indicates swapping and paging area(s), the “dumps” clause can be used to force system dumps to be taken on a particular device, and the “args” clause can be used to specify that argument list processing for *execve* should be done on a particular device.

The device names supplied in the clauses may be fully specified as a device, unit, and file system partition; or underspecified in which case *config* will use builtin rules to select default unit numbers and file system partitions. The defaulting rules are a bit complicated as they are dependent on the overall system configuration. For example, the swap area need not be specified at all if the root device is specified; in this case the swap area is placed in the “b” partition of the same disk where the root file system is located. Appendix B contains a complete list of the defaulting rules used in selecting system configuration devices.

The device names are translated to the appropriate major and minor device numbers on a per-machine basis. A file, “/sys/conf/devices.machine” (where “machine” is the machine type specified in the configuration file), is used to map a device name to its major block device number. The minor device number is calculated using the standard disk partitioning rules: on unit 0, partition “a” is minor device 0, partition “b” is minor device 1, and so on; for units other than 0, add 8 times the unit number to get the minor device.

If the default mapping of device name to major/minor device number is incorrect for your configuration, it can be replaced by an explicit specification of the major/minor device. This is done by substituting

```
major x minor y
```

where the device name would normally be found. For example,

```
config vmunix root on major 99 minor 1
```

Normally, the areas configured for swap space are sized by the system at boot time. If a non-standard size is to be used for one or more swap areas (less than the full partition), this can also be specified. To do this, the device name specified for a swap area should have a “size” specification appended. For example,

```
config vmunix root on hp0 swap on hp0b size 1200
```

would force swapping to be done in partition “b” of “hp0” and the swap partition size would be set to 1200 sectors. A swap area sized larger than the associated disk partition is trimmed to the partition size.

To create a generic configuration, only the clause “swap generic” should be specified; any extra clauses will cause an error.

#### 4.3. Device specifications

Each device attached to a machine must be specified to *config* so that the system generated will know to probe for it during the autoconfiguration process carried out at boot time. Hardware specified in the configuration need not actually be present on the machine where the generated system is to be run. Only the hardware actually found at boot time will be used by the system.

The specification of hardware devices in the configuration file parallels the interconnection hierarchy of the machine to be configured. On the VAX, this means that a configuration file must indicate what MASSBUS and UNIBUS adapters are present, and to which *next* they might be connected.\* Similarly, devices and controllers must be indicated as possibly being connected to one or

\* While VAX-11/750's and VAX-11/730 do not actually have *next*, the system treats them as having *simulated next* to simplify device configuration.

more adapters. A device description may provide a complete definition of the possible configuration parameters or it may leave certain parameters undefined and make the system probe for all the possible values. The latter allows a single device configuration list to match many possible physical configurations. For example, a disk may be indicated as present at UNIBUS adapter 0, or at any UNIBUS adapter which the system locates at boot time. The latter scheme, termed *wildcarding*, allows more flexibility in the physical configuration of a system; if a disk must be moved around for some reason, the system will still locate it at the alternate location.

A device specification takes one of the following forms:

```
master device-name device-info
controller device-name device-info [ interrupt-spec ]
device device-name device-info interrupt-spec
disk device-name device-info
tape device-name device-info
```

A “master” is a MASSBUS tape controller; a “controller” is a disk controller, a UNIBUS tape controller, a MASSBUS adapter, or a UNIBUS adapter. A “device” is an autonomous device which connects directly to a UNIBUS adapter (as opposed to something like a disk which connects through a disk controller). “Disk” and “tape” identify disk drives and tape drives connected to a “controller” or “master.”

The *device-name* is one of the standard device names, as indicated in section 4 of the UNIX Programmers Manual, concatenated with the *logical* unit number to be assigned the device (the *logical* unit number may be different than the *physical* unit number indicated on the front of something like a disk; the *logical* unit number is used to refer to the UNIX device, not the physical unit number). For example, “hp0” is logical unit 0 of a MASSBUS storage device, even though it might be physical unit 3 on MASSBUS adapter 1.

The *device-info* clause specifies how the hardware is connected in the interconnection hierarchy. On the VAX, UNIBUS and MASSBUS adapters are connected to the internal system bus through a *nexus*. Thus, one of the following specifications would be used:

```
controller      mba0      at nexus x
controller      uba0      at nexus x
```

To tie a controller to a specific nexus, “x” would be supplied as the number of that nexus; otherwise “x” may be specified as “?”, in which case the system will probe all nexi present looking for the specified controller.

The remaining interconnections on the VAX are:

- a controller may be connected to another controller (e.g. a disk controller attached to a UNIBUS adapter),
- a master is always attached to a controller (a MASSBUS adapter),
- a tape is always attached to a master (for MASSBUS tape drives),
- a disk is always attached to a controller, and
- devices are always attached to controllers (e.g. UNIBUS controllers attached to UNIBUS adapters).

The following lines give an example of each of these interconnections:

```
controller      hk0      at uba0 ...
master          ht0      at mba0 ...
disk            hp0      at mba0 ...
tape            tu0      at ht0 ...
disk            rk1      at hk0 ...
device          dz0      at uba0 ...
```

Any piece of hardware which may be connected to a specific controller may also be wildcarded across multiple controllers.

The final piece of information needed by the system to configure devices is some indication of where or how a device will interrupt. For tapes and disks, simply specifying the *slave* or *drive* number is sufficient to locate the control status register for the device. *Drive* numbers may be wild-carded on MASSBUS devices, but not on disks on a UNIBUS controller. For controllers, the control status register must be given explicitly, as well the number of interrupt vectors used and the names of the routines to which they should be bound. Thus the example lines given above might be completed as:

controller	hk0	at uba0 csr 0177440	vector rkintr
master	ht0	at mba0 drive 0	
disk	hp0	at mba0 drive ?	
tape	tu0	at ht0 slave 0	
disk	rk1	at hk0 drive 1	
device	dz0	at uba0 csr 0160100	vector dzrint dzxint

Certain device drivers require extra information passed to them at boot time to tailor their operation to the actual hardware present. The line printer driver, for example, needs to know how many columns are present on each non-standard line printer (i.e. a line printer with other than 80 columns). The drivers for the terminal multiplexors need to know which lines are attached to modem lines so that no one will be allowed to use them unless a connection is present. For this reason, one last parameter may be specified to a *device*, a *flags* field. It has the syntax

*flags number*

and is usually placed after the *csr* specification. The *number* is passed directly to the associated driver. The manual pages in section 4 should be consulted to determine how each driver uses this value (if at all). Communications interface drivers commonly use the flags to indicate whether modem control signals are in use.

The exact syntax for each specific device is given in the Synopsis section of its manual page in section 4 of the manual.

#### 4.4. Pseudo-devices

A number of drivers and software subsystems are treated like device drivers without any associated hardware. To include any of these pieces, a "pseudo-device" specification must be used. A specification for a pseudo device takes the form

*pseudo-device device-name [ howmany ]*

Examples of pseudo devices are *pty*, the pseudo terminal driver (where the optional *howmany* value indicates the number of pseudo terminals to configure, 32 default), and *loop*, the software loop-back network pseudo-interface. Other pseudo devices for the network include *imp* (required when a CSS or ACC *imp* is configured) and *ether* (used by the Address Resolution Protocol on 10 Mb/sec Ethernet). More information on configuring each of these can also be found in section 4 of the manual.

### 5. SAMPLE CONFIGURATION FILES

In this section we will consider how to configure a sample VAX-11/780 system on which the hardware can be reconfigured to guard against various hardware mishaps. We then study the rules needed to configure a VAX-11/750 to run in a networking environment.

#### 5.1. VAX-11/780 System

Our VAX-11/780 is configured with hardware recommended in the document "Hints on Configuring a VAX for 4.2BSD" (this is one of the high-end configurations). Table 1 lists the pertinent hardware to be configured.



Item	Vendor	Connection	Name	Reference
cpu	DEC		VAX780	
MASSBUS controller	Emulex	nexus ?	mba0	hp(4)
disk	Fujitsu	mba0	hp0	
disk	Fujitsu	mba0	hp1	
MASSBUS controller	Emulex	nexus ?	mba1	
disk	Fujitsu	mba1	hp2	tm(4)
disk	Fujitsu	mba1	hp3	
UNIBUS adapter	DEC	nexus ?		
tape controller	Emulex	uba0	tm0	
tape drive	Kennedy	tm0	te0	dh(4)
tape drive	Kennedy	tm0	te1	
terminal multiplexor	Emulex	uba0	dh0	
terminal multiplexor	Emulex	uba0	dh1	
terminal multiplexor	Emulex	uba0	dh2	

Table 1. VAX-11/780 Hardware support.

We will call this machine ANSEL and construct a configuration file one step at a time.

The first step is to fill in the global configuration parameters. The machine is a VAX, so the *machine type* is "vax". We will assume this system will run only on this one processor, so the *cpu type* is "VAX780". The options are empty since this is going to be a "vanilla" VAX. The system identifier, as mentioned before, is "ANSEL," and the maximum number of users we plan to support is about 40. Thus the beginning of the configuration file looks like this:

```
#
# ANSEL VAX (a picture perfect machine)
#
machine          vax
cpu              VAX780
timezone         8 dst
ident            ANSEL
maxusers         40
```

To this we must then add the specifications for three system images. The first will be our standard system with the root on "hp0" and swapping on the same drive as the root. The second will have the root file system in the same location, but swap space interleaved among drives on each controller. Finally, the third will be a generic system, to allow us to boot off any of the four disk drives.

```
config          vmunix          root on hp0
config          hpvmunix        root on hp0 swap on hp0 and hp2
config          genvmunix        swap generic
```

Finally, the hardware must be specified. Let us first just try transcribing the information from Table 1.





controller	mba0	at nexus ?	
disk	hp0	at mba0 disk 0	
disk	hp1	at mba0 disk 1	
controller	mba1	at nexus ?	
disk	hp2	at mba1 disk 2	
disk	hp3	at mba1 disk 3	
controller	uba0	at nexus ?	
controller	tm0	at uba0 csr 0172520	vector tmintr
tape	te0	at tm0 drive 0	
tape	te1	at tm0 drive 1	
device	dh0	at uba0 csr 0160020	vector dhrintr dhxint
device	dm0	at uba0 csr 0170500	vector dmintr
device	dh1	at uba0 csr 0160040	vector dhrintr dhxint
device	dh2	at uba0 csr 0160060	vector dhrintr dhxint

(Oh, I forgot to mention one panel of the terminal multiplexor has modem control, thus the "dm0" device.)

This will suffice, but leaves us with little flexibility. Suppose our first disk controller were to break. We would like to recable the drives normally on the second controller so that all our disks could still be used without reconfiguring the system. To do this we wildcard the MASSBUS adapter connections and also the slave numbers. Further, we wildcard the UNIBUS adapter connections in case we decide some time in the future to purchase another adapter to offload the single UNIBUS we currently have. The revised device specifications would then be:

controller	mba0	at nexus ?	
disk	hp0	at mba? disk ?	
disk	hp1	at mba? disk ?	
controller	mba1	at nexus ?	
disk	hp2	at mba? disk ?	
disk	hp3	at mba? disk ?	
controller	uba0	at nexus ?	
controller	tm0	at uba? csr 0172520	vector tmintr
tape	te0	at tm0 drive 0	
tape	te1	at tm0 drive 1	
device	dh0	at uba? csr 0160020	vector dhrintr dhxint
device	dm0	at uba? csr 0170500	vector dmintr
device	dh1	at uba? csr 0160040	vector dhrintr dhxint
device	dh2	at uba? csr 0160060	vector dhrintr dhxint

The completed configuration file for ANSEL is shown in Appendix C.

## 5.2. VAX-11/750 with network support

Our VAX-11/750 system will be located on two 10Mb/s Ethernet local area networks and also the DARPA Internet. The system will have a MASSBUS drive for the root file system and two UNIBUS drives. Paging is interleaved among all three drives. We have sold our standard DEC terminal multiplexors since this machine will be accessed solely through the network. This machine is not intended to have a large user community, it does not have a great deal of memory. First the global parameters:



```
#
# UCBVAX (Gateway to the world)
#
machine      vax
cpu          "VAX780"
cpu          "VAX750"
ident        UCBVAX
timezone     8 dst
maxusers     32
options      INET
options      NS
```

The multiple cpu types allow us to replace UCBVAX with a more powerful cpu without reconfiguring the system. The value of 32 given for the maximum number of users is done to force the system data structures to be over-allocated. That is desirable on this machine because, while it is not expected to support many users, it is expected to perform a great deal of work. The "INET" indicates that we plan to use the DARPA standard Internet protocols on this machine, and "NS" also includes support for Xerox NS protocols. Note that unlike 4.2BSD configuration files, the network protocol options do not require corresponding pseudo devices.

The system images and disks are configured next.

config	vmunix	root on hp swap on hp and rk0 and rk1
config	upvmunix	root on up
config	hkvmutex	root on hk swap on rk0 and rk1
controller	mba0	at nexus ?
controller	uba0	at nexus ?
disk	hp0	at mba? drive 0
disk	hp1	at mba? drive 1
controller	sc0	at uba? csr 0176700      vector upintr
disk	up0	at sc0 drive 0
disk	up1	at sc0 drive 1
controller	hk0	at uha? csr 0177440      vector rkintr
disk	rk0	at hk0 drive 0
disk	rk1	at hk0 drive 1

UCBVAX requires heavy interleaving of its paging area to keep up with all the mail traffic it handles. The limiting factor on this system's performance is usually the number of disk arms, as opposed to memory or cpu cycles. The extra UNIBUS controller, "sc0", is in case the MASSBUS controller breaks and a spare controller must be installed (most of our old UNIBUS controllers have been replaced with the newer MASSBUS controllers, so we have a number of these around as spares).

Finally, we add in the network devices. Pseudo terminals are needed to allow users to log in across the network (remember the only hardwired terminal is the console). The software loopback device is used for on-machine communications. The connection to the Internet is through an IMP, this requires yet another *pseudo-device* (in addition to the actual hardware device used by the IMP software). And, finally, there are the two Ethernet devices. These use a special protocol, the Address Resolution Protocol (ARP), to map between Internet and Ethernet addresses. Thus, yet another *pseudo-device* is needed. The additional device specifications are shown below.



pseudo-device	pty		
pseudo-device	loop		
pseudo-device	imp		
device	acc0	at uba? csr 0167600	vector accrint accxint
pseudo-device	ether		
device	ec0	at uba? csr 0164330	vector ecrint eccollide ecxint
device	ilo	at uba? csr 0164000	vector ilrint ilcint

The completed configuration file for UCBVAX is shown in Appendix C.

### 5.3. Miscellaneous comments

It should be noted in these examples that neither system was configured to use disk quotas or the 4.1BSD compatibility mode. To use these optional facilities, and others, we would probably clean out our current configuration, reconfigure the system, then recompile and relink the system image(s). This could, of course, be avoided by figuring out which relocatable object files are affected by the reconfiguration, then reconfiguring and recompiling only those files affected by the configuration change. This technique should be used carefully.

## 6. ADDING NEW SYSTEM SOFTWARE

This section is not for the novice, it describes some of the inner workings of the configuration process as well as the pertinent parts of the system autoconfiguration process. It is intended to give those people who intend to install new device drivers and/or other system facilities sufficient information to do so in the manner which will allow others to easily share the changes.

This section is broken into four parts:

- general guidelines to be followed in modifying system code,
- how to add non-standard system facilities to 4.3BSD,
- how to add a device driver to 4.3BSD, and
- how UNIBUS device drivers are autoconfigured under 4.3BSD on the VAX.

### 6.1. Modifying system code

If you wish to make site-specific modifications to the system it is best to bracket them with

```
#ifdef SITENAME
...
#endif
```

to allow your source to be easily distributed to others, and also to simplify *diff(1)* listings. If you choose not to use a source code control system (e.g. SCCS, RCS), and perhaps even if you do, it is recommended that you save the old code with something of the form:

```
#ifndef SITENAME
...
#endif
```

We try to isolate our site-dependent code in individual files which may be configured with pseudo-device specifications.

Indicate machine-specific code with “`#ifdef vax`” (or other machine, as appropriate). 4.2BSD underwent extensive work to make it extremely portable to machines with similar architectures— you may someday find yourself trying to use a single copy of the source code on multiple machines.

Use *lint* periodically if you make changes to the system. The 4.3BSD kernel has only two lines of *lint* in it. It is very simple to lint the kernel. Use the LINT configuration file, designed to pull in as much of the kernel source code as possible, in the following manner.

```

$ cd /sys/conf
$ mkdir ../LINT
$ config LINT
$ cd ../LINT
$ make depend
$ make assym.s
$ make -k lint > linterrs 2>&1 &
(or for users of csh(1))
% make -k >& linterrs

```

This takes about an hour on a lightly loaded VAX-11/750, but is well worth it.

## 6.2. Adding non-standard system facilities

This section considers the work needed to augment *config*'s data base files for non-standard system facilities. *Config* uses a set of files that list the source modules that may be required when building a system. The data bases are taken from the directory in which *config* is run, normally */sys/conf*. Three such files may be used: *files*, *files.machine*, and *files.ident*. The first is common to all systems, the second contains files unique to a single machine type, and the third is an optional list of modules for use on a specific machine. This last file may override specifications in the first two. The format of the *files* file has grown somewhat complex over time. Entries are normally of the form

```
dir/source.c    type    option-list modifiers
```

for example,

```
vaxuba/foo.c    optional    foo    device-driver
```

The *type* is one of *standard* or *Files* marked as *standard* are included in all system configurations. Optional file specifications include a list of one or more system options that together require the inclusion of this module. The options in the list may be either names of devices that may be in the configuration file, or the names of system options that may be defined. An optional file may be listed multiple times with different options; if all of the options for any of the entries are satisfied, the module is included.

If a file is specified as a *device-driver*, any special compilation options for device drivers will be invoked. On the VAX this results in the use of the *-i* option for the C optimizer. This is required when pointer references are made to memory locations in the VAX I/O address space.

Two other optional keywords modify the usage of the file. *Config* understands that certain files are used especially for kernel profiling. These files are indicated in the *files* files with a *profiling-routine* keyword. For example, the current profiling subroutines are sequestered off in a separate file with the following entry:

```
sys/subr_mcount.c    optional    profiling-routine
```

The *profiling-routine* keyword forces *config* not to compile the source file with the *-pg* option.

The second keyword which can be of use is the *config-dependent* keyword. This causes *config* to compile the indicated module with the global configuration parameters. This allows certain modules, such as *machdep.c* to size system data structures based on the maximum number of users configured for the system.

## 6.3. Adding device drivers to 4.3BSD

The I/O system and *config* have been designed to easily allow new device support to be added. The system source directories are organized as follows:



/sys/h	machine independent include files
/sys/sys	machine-independent system source files
/sys/conf	site configuration files and basic templates
/sys/net	network-protocol-independent, but network-related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netns	Xerox NS code
/sys/vax	VAX-specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Existing block and character device drivers for the VAX reside in `"/sys/vax"`, `"/sys/vaxmba"`, and `"/sys/vaxuba"`. Network interface drivers reside in `"/sys/vaxif"`. Any new device drivers should be placed in the appropriate source code directory and named so as not to conflict with existing devices. Normally, definitions for things like device registers are placed in a separate file in the same directory. For example, the `"dh"` device driver is named `"dh.c"` and its associated include file is named `"dhreg.h"`.

Once the source for the device driver has been placed in a directory, the file `"/sys/conf/files.machine"`, and possibly `"/sys/conf/devices.machine"` should be modified. The *files* files in the `conf` directory contain a line for each C source or binary-only file in the system. Those files which are machine independent are located in `"/sys/conf/files,"` while machine specific files are in `"/sys/conf/files.machine."` The `"devices.machine"` file is used to map device names to major block device numbers. If the device driver being added provides support for a new disk you will want to modify this file (the format is obvious).

In addition to including the driver in the *files* file, it must also be added to the device configuration tables. These are located in `"/sys/vax/conf.c"`, or similar for machines other than the VAX. If you don't understand what to add to this file, you should study an entry for an existing driver. Remember that the position in the device table specifies the major device number. The block major number is needed in the `"devices.machine"` file if the device is a disk.

With the configuration information in place, your configuration file appropriately modified, and a system reconfigured and rebooted you should incorporate the shell commands needed to install the special files in the file system to the file `"/dev/MAKEDEV"` or `"/dev/MAKEDEV.local"`. This is discussed in the document "Installing and Operating 4.3BSD on the VAX".

#### 6.4. Autoconfiguration on the VAX

4.3BSD requires all device drivers to conform to a set of rules which allow the system to:

- 1) support multiple UNIBUS and MASSBUS adapters,
- 2) support system configuration at boot time, and
- 3) manage resources so as not to crash when devices request resources which are unavailable.

In addition, devices such as the RK07 which require everyone else to get off the UNIBUS when they are running need cooperation from other DMA devices if they are to work. Since it is unlikely that you will be writing a device driver for a MASSBUS device, this section is devoted exclusively to describing the I/O system and autoconfiguration process as it applies to UNIBUS devices.

Each UNIBUS on a VAX has a set of resources:

- 496 map registers which are used to convert from the 18-bit UNIBUS addresses into the much larger VAX memory address space.
- Some number of buffered data paths (3 on an 11/750, 15 on an 11/780, 0 on an 11/730) which are used by high speed devices to transfer data using fewer bus cycles.



There is a structure of type *struct uba\_hd* in the system per UNIBUS adapter used to manage these resources. This structure also contains a linked list where devices waiting for resources to complete DMA UNIBUS activity have requests waiting.

There are three central structures in the writing of drivers for UNIBUS controllers; devices which do not do DMA I/O can often use only two of these structures. The structures are *struct uba\_ctlr*, the UNIBUS controller structure, *struct uba\_device* the UNIBUS device structure, and *struct uba\_driver*, the UNIBUS driver structure. The *uba\_ctlr* and *uba\_device* structures are in one-to-one correspondence with the definitions of controllers and devices in the system configuration. Each driver has a *struct uba\_driver* structure specifying an internal interface to the rest of the system.

Thus a specification

controller sc0 at uba0 csr 0176700 vector upintr

would cause a *struct uba\_ctlr* to be declared and initialized in the file *ioconf.c* for the system configured from this description. Similarly specifying

disk up0 at sc0 drive 0

would declare a related *uba\_device* in the same file. The *up.c* driver which implements this driver specifies in its declarations:

```
int  upprobe(), upslave(), upattach(), updgo(), upintr();
struct uba_ctlr *upminfo[NSC];
struct uba_device *updinfo[NUP];
u_short upstd[] = { 0776700, 0774400, 0776300, 0 };
struct uba_driver scdriver =
    { upprobe, upslave, upattach, updgo, upstd, "up", updinfo, "sc", upminfo };
```

initializing the *uba\_driver* structure. The driver will support some number of controllers named *sc0*, *sc1*, etc, and some number of drives named *up0*, *up1*, etc. where the drives may be on any of the controllers (that is there is a single linear name space for devices, separate from the controllers.)

We now explain the fields in the various structures. It may help to look at a copy of *vaxuba/ubareg.h*, *vaxuba/ubavar.h* and drivers such as *up.c* and *dz.c* while reading the descriptions of the various structure fields.

#### **uba\_driver structure**

One of these structures exists per driver. It is initialized in the driver and contains functions used by the configuration program and by the UNIBUS resource routines. The fields of the structure are:

##### **ud\_probe**

A routine which, given a *caddr\_t* address as argument, should attempt to determine that the device is present at that address in virtual memory, and should cause an interrupt from the device. When probing controllers, two additional arguments are supplied: the controller index, and a pointer to the *uba\_ctlr* structure. Device probe routines receive a pointer to the *uba\_device* structure as second argument. Both of these structures are described below. Neither is normally used, but devices that must record status or device type information from the probe routine may require them.

The autoconfiguration routine attempts to verify that the specified address responds before calling the probe routine. However, the device may not actually exist or may be of a different type, and therefore the probe routine should use delays (via the *DELAY(n)* macro which delays for *n* microseconds) rather than waiting for specific events to occur. The routine must not declare its argument as a *register* parameter, but must declare

```
register int br, cvec;
```

as local variables. At boot time the system takes special measures that these variables are "value-result" parameters. The *br* is the IPL of the device when it interrupts, and the *cvec* is the interrupt

vector address on the UNIBUS. These registers are actually filled in in the interrupt handler when an interrupt occurs.

As an example, here is the *up.c* probe routine:

```
upprobe(reg)
    caddr_t reg;
    {
        register int br, cvec;

#ifdef lint
        br = 0; cvec = br; br = cvec; upintr(0);
#endif
        ((struct updevice *)reg)->upcs1 = UP_IE|UP_RDY;
        DELAY(10);
        ((struct updevice *)reg)->upcs1 = 0;
        return (sizeof (struct updevice));
    }
```

The definitions for *lint* serve to indicate to it that the *br* and *cvec* variables are value-result. The call to the interrupt routine satisfies *lint* that the interrupt handler is used. The code here enables interrupts on the device and writes the ready bit *UP\_RDY*. The 10 microsecond delay insures that the interrupt enable will not be canceled before the interrupt can be posted. The return of "sizeof (struct updevice)" here indicates that the probe routine is satisfied that the device is present (the value returned is not currently used, but future plans dictate that you should return the amount of space in the device's register bank). A probe routine may use the function "badaddr" to see if certain other addresses are accessible on the UNIBUS (without generating a machine check), or look at the contents of locations where certain registers should be. If the registers' contents are not acceptable or the addresses don't respond, the probe routine can return 0 and the device will not be considered to be there.

One other thing to note is that the action of different VAXen when illegal addresses are accessed on the UNIBUS may differ. Some of the machines may generate machine checks and some may cause UNIBUS errors. Such considerations are handled by the configuration program and the driver writer need not be concerned with them.

It is also possible to write a very simple probe routine for a one-of-a-kind device if probing is difficult or impossible. Such a routine would include statements of the form:

```
br = 0x15;
cvec = 0200;
```

for instance, to declare that the device ran at UNIBUS *br5* and interrupted through vector *0200* on the UNIBUS.

*ud\_slave*

This routine is called with a *uba\_device* structure (yet to be described) and the address of the device controller. It should determine whether a particular slave device of a controller is present, returning 1 if it is and 0 if it is not. As an example here is the slave routine for *up.c*.





```

upslave(ui, reg)
    struct uba_device *ui;
    caddr_t reg;
{
    register struct updevice *upaddr = (struct updevice *)reg;

    upaddr->upcs1 = 0;          /* conservative */
    upaddr->upcs2 = ui->ui_slave;
    if (upaddr->upcs2 & UPCS2_NED) {
        upaddr->upcs1 = UP_DCLR | UP_GO;
        return (0);
    }
    return (1);
}

```

Here the code fetches the slave (disk unit) number from the *ui\_slave* field of the *uba\_device* structure, and sees if the controller responds that that is a non-existent driver (NED). If the drive is not present, a drive clear is issued to clean the state of the controller, and 0 is returned indicating that the slave is not there. Otherwise a 1 is returned.

#### ud\_attach

The attach routine is called after the autoconfigure code and the driver concur that a peripheral exists attached to a controller. This is the routine where internal driver state about the peripheral can be initialized. Here is the *attach* routine from the *up.c* driver:

```

upattach(ui)
    register struct uba_device *ui;
{
    register struct updevice *upaddr;

    if (upwstart == 0) {
        timeout(upwatch, (caddr_t)0, hz);
        upwstart++;
    }
    if (ui->ui_dk >= 0)
        dk_mspw[ui->ui_dk] = .0000020345;
    upip[ui->ui_ctlr][ui->ui_slave] = ui;
    up_softc[ui->ui_ctlr].sc_ndrive++;
    ui->ui_type = upmaptype(ui);
}

```

The attach routine here performs a number of functions. The first time any drive is attached to the controller it starts the timeout routine which watches the disk drives to make sure that interrupts aren't lost. It also initializes, for devices which have been assigned *iostat* numbers (when *ui->ui\_dk* >= 0), the transfer rate of the device in the array *dk\_mspw*, the fraction of a second it takes to transfer 16 bit word. It then initializes an inverting pointer in the array *upip* which will be used later to determine, for a particular *up* controller and slave number, the corresponding *uba\_device*. It increments the count of the number of devices on this controller, so that search commands can later be avoided if the count is exactly 1. It then attempts to decipher the actual type of drive attached to the controller in a controller-specific way. On the EMULEX SC-21 it may ask for the number of tracks on the device and use this to decide what the drive type is. The drive type is used to setup disk partition mapping tables and other device specific information.

#### ud\_dgo

This is the routine which is called by the UNIBUS resource management routines when an operation is ready to be started (because the required resources have been allocated). The

routine in *up.c* is:

```

updgo(um)
    struct uba_ctlr *um;
{
    register struct updevice *upaddr = (struct updevice *)um->um_addr;

    upaddr->upba = um->um_ubinfo;
    upaddr->upcs1 = um->um_cmd|((um->um_ubinfo>>8)&0x300);
}

```

This routine uses the field *um\_ubinfo* of the *uba\_ctlr* structure which is where the UNIBUS routines store the UNIBUS map allocation information. In particular, the low 18 bits of this word give the UNIBUS address assigned to the transfer. The assignment to *upba* in the *go* routine places the low 16 bits of the UNIBUS address in the disk UNIBUS address register. The next assignment places the disk operation command and the extended (high 2) address bits in the device control-status register, starting the I/O operation. The field *um\_cmd* was initialized with the command to be stuffed here in the driver code itself before the call to the *ubago* routine which eventually resulted in the call to *updgo*.

#### **ud\_addr**

This is a zero-terminated list of the conventional addresses for the device control registers in UNIBUS space. This information is used by the system to look for instances of the device supported by the driver. When the system probes for the device it first checks for a control-status register located at the address indicated in the configuration file (if supplied), then uses the list of conventional addresses pointed to be *ud\_addr*.

#### **ud\_dname**

This is the name of a *device* supported by this controller; thus the disks on a SC-21 controller are called *up0*, *up1*, etc. That is because this field contains *up*.

#### **ud\_dinfo**

This is an array of back pointers to the *uba\_device* structures for each device attached to the controller. Each driver defines a set of controllers and a set of devices. The device address space is always one-dimensional, so that the presence of extra controllers may be masked away (e.g. by pattern matching) to take advantage of hardware redundancy. This field is filled in by the configuration program, and used by the driver.

#### **ud\_mname**

The name of a controller, e.g. *sc* for the *up.c* driver. The first SC-21 is called *sc0*, etc.

#### **ud\_minfo**

The backpointer array to the structures for the controllers.

#### **ud\_xclu**

If non-zero specifies that the controller requires exclusive use of the UNIBUS when it is running. This is non-zero currently only for the RK611 controller for the RK07 disks to map around a hardware problem. It could also be used if 6250bpi tape drives are to be used on the UNIBUS to insure that they get the bandwidth that they need (basically the whole bus).

#### **ud\_ubamem**

This is an optional entry point to the driver to configure UNIBUS memory associated with a device. If this field in the driver structure is null, it is ignored. Otherwise, it is called before beginning to probe for devices when configuration of a UNIBUS is begun. The driver must probe for the existence of its memory, and is then responsible for allocating the map registers corresponding to the device memory addresses so that the registers are not used for other purposes. The *ud\_ubamem* returns 0 on success and -1 on failure. A return value of 1 indicates that the memory exists, and that there is no further configuration required for the device.

**uba\_ctlr structure**

One of these structures exists per-controller. The fields link the controller to its UNIBUS adapter and contain the state information about the devices on the controller. The fields are:

**um\_driver**

A pointer to the *struct uba\_device* for this driver, which has fields as defined above.

**um\_ctlr**

The controller number for this controller, e.g. the 0 in *sc0*.

**um\_alive**

Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba\_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

**um\_intr**

The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

**um\_hd**

A back-pointer to the UNIBUS adapter to which this controller is attached.

**um\_cmd**

A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba\_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the I/O operation.

**um\_ubinfo**

Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

**um\_tab**

This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a *buf* structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the *buf* structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:

- A chain of *buf* structures for each waiting device on this controller.
- On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the I/O operation.

**uba\_device structure**

One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA I/O may have only a device structure. Thus *dz* and *dh* devices have only *uba\_device* structures. The fields are:

**ui\_driver**

A pointer to the *struct uba\_driver* structure for this device type.

**ui\_unit**

The unit number of this device, e.g. 0 in *up0*, or 1 in *dh1*.

**ui\_ctlr**

The number of the controller on which this device is attached, or -1 if this device is not on a controller.

**ui\_ubanum**

The number of the UNIBUS on which this device is attached.

**ui\_slave**

The slave number of this device on the controller which it is attached to, or -1 if the device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui\_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

**ui\_intr**

The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s*.

**ui\_addr**

The control-status register address of this device.

**ui\_dk**

The iostat number assigned to this device. Numbers are assigned to disks only, and are small nonnegative integers which index the various *dk\_\** arrays in *<sys/dk.h>*.

**ui\_flags**

The optional "flags xxx" parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

**ui\_alive**

The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

**ui\_type**

The device type, to be used by the driver internally.

**ui\_physaddr**

The physical memory address of the device control-status register. This is typically used in the device dump routines.

**ui\_mi**

A *struct uba\_ctlr* pointer to the controller (if any) on which this device resides.

**ui\_hd**

A *struct uba\_hd* pointer to the UNIBUS on which this device resides.

**UNIBUS resource management routines**

UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

**NEEDBDP**

if you need a buffered data path,

**HAVEBDP**

if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS),

**CANTWAIT**

if you are calling (potentially) from interrupt level, and

**NEED16**

if the device uses only 16 address bits, and thus requires map registers from the first 64K of UNIBUS address space.

If the presentation here does not answer all the questions you may have, consult the file */sys/vaxuba/uba.c*

### Autoconfiguration requirements

Basically all you have to do is write a *ud\_probe* and a *ud\_attach* routine for the controller. It suffices to have a *ud\_probe* routine which just initializes *br* and *cvec*, and a *ud\_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don't be surprised at undefined structure complaints if you order the includes incorrectly. Finally, if you get the device configured in, you can try bootstrapping and see if configuration messages print out about your device. It is a good idea to have some messages in the probe routine so that you can see that it is being called and what is going on. If it is not called, then you probably have the control-status register address wrong in the system configuration. The autoconfigure code notices that the device doesn't exist in this case, and the probe will never be called.

Assuming that your probe routine works and you manage to generate an interrupt, then you are basically back to where you would have been under older versions of UNIX. Just be sure to use the *ui\_ctlr* field of the *uba\_device* structures to address the device; compiling in funny constants will make your driver only work on the CPU type you have (780, 750, or 730).

Other bad things that might happen while you are setting up the configuration stuff:

- You get "nexus zero vector" errors from the system. This will happen if you cause a device to interrupt, but take away the interrupt enable so fast that the UNIBUS adapter cancels the interrupt and confuses the processor. The best thing to do it to put a modest delay in the probe code between the instructions which should cause an interrupt and the clearing of the interrupt enable. (You should clear interrupt enable before you leave the probe routine so the device doesn't interrupt more and confuse the system while it is configuring other devices.)
- The device refuses to interrupt or interrupts with a "zero vector". This typically indicates a problem with the hardware or, for devices which emulate other devices, that the emulation is incomplete. Devices may fail to present interrupt vectors because they have configuration switches set wrong, or because they are being accessed in inappropriate ways. Incomplete emulation can cause "maintenance mode" features to not work properly, and these features are often needed to force device interrupts.

## APPENDIX A. CONFIGURATION FILE GRAMMAR

S  
M  
M  
2

The following grammar is a compressed form of the actual *yacc*(1) grammar used by *config* to parse configuration files. Terminal symbols are shown all in upper case, literals are emboldened; optional clauses are enclosed in brackets, “[” and “]”; zero or more instantiations are denoted with “\*”.

```

Configuration ::= [ Spec ; ]*

Spec ::= Config_spec
      | Device_spec
      | trace
      | /* lambda */

/* configuration specifications */

Config_spec ::= machine ID
            | cpu ID
            | options Opt_list
            | ident ID
            | System_spec
            | timezone [ - ] NUMBER [ dst [ NUMBER ] ]
            | timezone [ - ] FPNUMBER [ dst [ NUMBER ] ]
            | maxusers NUMBER

/* system configuration specifications */

System_spec ::= config ID System_parameter [ System_parameter ]*

System_parameter ::= swap_spec | root_spec | dump_spec | arg_spec

swap_spec ::= swap [ on ] swap_dev [ and swap_dev ]*

swap_dev ::= dev_spec [ size NUMBER ]

root_spec ::= root [ on ] dev_spec

dump_spec ::= dumps [ on ] dev_spec

arg_spec ::= args [ on ] dev_spec

dev_spec ::= dev_name | major_minor

major_minor ::= major NUMBER minor NUMBER

dev_name ::= ID [ NUMBER [ ID ] ]

/* option specifications */

Opt_list ::= Option [ , Option ]*

Option ::= ID [ = Opt_value ]

Opt_value ::= ID | NUMBER

```



```

Mkopt_list ::= Mkoption [ , Mkoption ]*

Mkoption ::= ID = Opt_value

/* device specifications */

Device_spec ::= device Dev_name Dev_info Int_spec
    | master Dev_name Dev_info
    | disk Dev_name Dev_info
    | tape Dev_name Dev_info
    | controller Dev_name Dev_info [ Int_spec ]
    | pseudo-device Dev [ NUMBER ]

Dev_name ::= Dev NUMBER

Dev ::= uba | mba | ID

Dev_info ::= Con_info [ Info ]*

Con_info ::= at Dev NUMBER
    | at nexus NUMBER

Info ::= csr NUMBER
    | drive NUMBER
    | slave NUMBER
    | flags NUMBER

Int_spec ::= vector ID [ ID ]*
    | priority NUMBER

```

### Lexical Conventions

The terminal symbols are loosely defined as:

#### ID

One or more alphabets, either upper or lower case, and underscore, “\_”.

#### NUMBER

Approximately the C language specification for an integer number. That is, a leading “0x” indicates a hexadecimal value, a leading “0” indicates an octal value, otherwise the number is expected to be a decimal value. Hexadecimal numbers may use either upper or lower case alphabets.

#### FPNUMBER

A floating point number without exponent. That is a number of the form “nnn.ddd”, where the fractional component is optional.

In special instances a question mark, “?”, can be substituted for a “NUMBER” token. This is used to effect wildcarding in device interconnection specifications.

Comments in configuration files are indicated by a “#” character at the beginning of the line; the remainder of the line is discarded.

A specification is interpreted as a continuation of the previous line if the first character of the line is tab.

## APPENDIX B. RULES FOR DEFAULTING SYSTEM DEVICES

When *config* processes a “config” rule which does not fully specify the location of the root file system, paging area(s), device for system dumps, and device for argument list processing it applies a set of rules to define those values left unspecified. The following list of rules are used in defaulting system devices.

- 1) If a root device is not specified, the swap specification must indicate a “generic” system is to be built.
- 2) If the root device does not specify a unit number, it defaults to unit 0.
- 3) If the root device does not include a partition specification, it defaults to the “a” partition.
- 4) If no swap area is specified, it defaults to the “b” partition of the root device.
- 5) If no device is specified for processing argument lists, the first swap partition is selected.
- 6) If no device is chosen for system dumps, the first swap partition is selected (see below to find out where dumps are placed within the partition).

The following table summarizes the default partitions selected when a device specification is incomplete, e.g. “hp0”.

Type	Partition
root	“a”
swap	“b”
args	“b”
dumps	“b”

### Multiple swap/paging areas

When multiple swap partitions are specified, the system treats the first specified as a “primary” swap area which is always used. The remaining partitions are then interleaved into the paging system at the time a *swapon*(2) system call is made. This is normally done at boot time with a call to *swapon*(8) from the */etc/rc* file.

### System dumps

System dumps are automatically taken after a system crash, provided the device driver for the “dumps” device supports this. The dump contains the contents of memory, but not the swap areas. Normally the dump device is a disk in which case the information is copied to a location at the back of the partition. The dump is placed in the back of the partition because the primary swap and dump device are commonly the same device and this allows the system to be rebooted without immediately overwriting the saved information. When a dump has occurred, the system variable *dumpsiz*e is set to a non-zero value indicating the size (in bytes) of the dump. The *savecore*(8) program then copies the information from the dump partition to a file in a “crash” directory and also makes a copy of the system which was running at the time of the crash (usually “/vmunix”). The offset to the system dump is defined in the system variable *dumplo* (a sector offset from the front of the dump partition). The *savecore* program operates by reading the contents of *dumplo*, *dumpdev*, and *dumpmagic* from */dev/kmem*, then comparing the value of *dumpmagic* read from */dev/kmem* to that located in corresponding location in the dump area of the dump partition. If a match is found, *savecore* assumes a crash occurred and reads *dumpsiz*e from the dump area of the dump partition. This value is then used in copying the system dump. Refer to *savecore*(8) for more information about its operation.

The value *dumplo* is calculated to be

$$\text{dumpdev-size} - \text{memsize}$$

where *dumpdev-size* is the size of the disk partition where system dumps are to be placed, and *memsize* is the size of physical memory. If the disk partition is not large enough to hold a full dump, *dumplo* is set to 0 (the start of the partition).



## APPENDIX C. SAMPLE CONFIGURATION FILES

The following configuration files are developed in section 5; they are included here for completeness.

```
#
# ANSEL VAX (a picture perfect machine)
#
machine          vax
cpu              VAX780
timezone         8 dst
ident            ANSEL
maxusers         40

config           vmunix      root on hp0
config           hpvmunix    root on hp0 swap on hp0 and hp2
config           genvmunix   swap generic

controller       mba0        at nexus ?
disk             hp0         at mba? disk ?
disk             hp1         at mba? disk ?
controller       mba1        at nexus ?
disk             hp2         at mba? disk ?
disk             hp3         at mba? disk ?
controller       uba0        at nexus ?
controller       tm0         at uba? csr 0172520   vector tmintr
tape             te0         at tm0 drive 0
tape             te1         at tm0 drive 1
device           dh0         at uba? csr 0160020   vector dhrint dhxint
device           dm0         at uba? csr 0170500   vector dmintr
device           dh1         at uba? csr 0160040   vector dhrint dhxint
device           dh2         at uba? csr 0160060   vector dhrint dhxint
```



```

#
# UCBVAX - Gateway to the world
#
machine          vax
cpu              "VAX780"
cpu              "VAX750"
ident            UCBVAX
timezone         8 dst
maxusers         32
options          INET
options          NS

config           vmunix      root on hp swap on hp and rk0 and rk1
config           upvmunix   root on up
config           hkvmunix   root on hk swap on rk0 and rk1

controller       mba0       at nexus ?
controller       uba0       at nexus ?
disk             hp0        at mba? drive 0
disk             hp1        at mba? drive 1
controller       sc0        at uba? csr 0176700   vector upintr
disk             up0        at sc0 drive 0
disk             up1        at sc0 drive 1
controller       hk0        at uba? csr 0177440   vector rkintr
disk             rk0        at hk0 drive 0
disk             rk1        at hk0 drive 1
pseudo-device    pty
pseudo-device    loop
pseudo-device    imp
device           acc0       at uba? csr 0167600   vector accrint accxint
pseudo-device    ether
device           ec0        at uba? csr 0164330   vector ecrint eccollide ecxint
device           ilo        at uba? csr 0164000   vector ilrint ilcint

```

## APPENDIX D. VAX KERNEL DATA STRUCTURE SIZING RULES

Certain system data structures are sized at compile time according to the maximum number of simultaneous users expected, while others are calculated at boot time based on the physical resources present, e.g. memory. This appendix lists both sets of rules and also includes some hints on changing built-in limitations on certain data structures.



### Compile time rules

The file `/sys/conf/param.c` contains the definitions of almost all data structures sized at compile time. This file is copied into the directory of each configured system to allow configuration-dependent rules and values to be maintained. (Each copy normally depends on the copy in `/sys/conf`, and global modifications cause the file to be recopied unless the makefile is modified.) The rules implied by its contents are summarized below (here MAXUSERS refers to the value defined in the configuration file in the “maxusers” rule). Most limits are computed at compile time and stored in global variables for use by other modules; they may generally be patched in the system binary image before rebooting to test new values.

#### nproc

The maximum number of processes which may be running at any time. It is referred to in other calculations as NPROC and is defined to be

$$20 + 8 * \text{MAXUSERS}$$

#### ntext

The maximum number of active shared text segments. The constant is intended to allow for network servers and common commands that remain in the table. It is defined as

$$36 + \text{MAXUSERS}.$$

#### ninode

The maximum number of files in the file system which may be active at any time. This includes files in use by users, as well as directory files being read or written by the system and files associated with bound sockets in the UNIX IPC domain. It is defined as

$$(\text{NPROC} + 16 + \text{MAXUSERS}) + 32$$

#### nfile

The number of “file table” structures. One file table structure is used for each open, unshared, file descriptor. Multiple file descriptors may reference a single file table entry when they are created through a *dup* call, or as the result of a *fork*. This is defined to be

$$16 * (\text{NPROC} + 16 + \text{MAXUSERS}) / 10 + 32$$

#### ncallout

The number of “callout” structures. One callout structure is used per internal system event handled with a timeout. Timeouts are used for terminal delays, watchdog routines in device drivers, protocol timeout processing, etc. This is defined as

$$16 + \text{NPROC}$$

#### nclist

The number of “c-list” structures. C-list structures are used in terminal I/O, and currently each holds 60 characters. Their number is defined as

$$60 + 12 * \text{MAXUSERS}$$

#### nmbclusters

The maximum number of pages which may be allocated by the network. This is defined as 256 (a quarter megabyte of memory) in `/sys/h/mbuf.h`. In practice, the network rarely uses this

much memory. It starts off by allocating 8 kilobytes of memory, then requesting more as required. This value represents an upper bound.

nquota

The number of "quota" structures allocated. Quota structures are present only when disc quotas are configured in the system. One quota structure is kept per user. This is defined to be

$$(MAXUSERS * 9) / 7 + 3$$

ndquot

The number of "dquot" structures allocated. Dquot structures are present only when disc quotas are configured in the system. One dquot structure is required per user, per active file system quota. That is, when a user manipulates a file on a file system on which quotas are enabled, the information regarding the user's quotas on that file system must be in-core. This information is cached, so that not all information must be present in-core all the time. This is defined as

$$NINODE + (MAXUSERS * NMOUNT) / 4$$

where NMOUNT is the maximum number of mountable file systems.

In addition to the above values, the system page tables (used to map virtual memory in the kernel's address space) are sized at compile time by the SYSPTSIZE definition in the file /sys/vax/vmparam.h. This is defined to be

$$20 + MAXUSERS$$

pages of page tables. Its definition affects the size of many data structures allocated at boot time because it constrains the amount of virtual memory which may be addressed by the running system. This is often the limiting factor in the size of the buffer cache, in which case a message is printed when the system configures at boot time.

#### Run-time calculations

The most important data structures sized at run-time are those used in the buffer cache. Allocation is done by allocating physical memory (and system virtual memory) immediately after the system has been started up; look in the file /sys/vax/machdep.c. The amount of physical memory which may be allocated to the buffer cache is constrained by the size of the system page tables, among other things. While the system may calculate a large amount of memory to be allocated to the buffer cache, if the system page table is too small to map this physical memory into the virtual address space of the system, only as much as can be mapped will be used.

The buffer cache is comprised of a number of "buffer headers" and a pool of pages attached to these headers. Buffer headers are divided into two categories: those used for swapping and paging, and those used for normal file I/O. The system tries to allocate 10% of the first two megabytes and 5% of the remaining available physical memory for the buffer cache (where *available* does not count that space occupied by the system's text and data segments). If this results in fewer than 16 pages of memory allocated, then 16 pages are allocated. This value is kept in the initialized variable *bufpages* so that it may be patched in the binary image (to allow tuning without recompiling the system), or the default may be overridden with a configuration-file option. For example, the option `options BUF-PAGES="3200"` causes 3200 pages (3.2M bytes) to be used by the buffer cache. A sufficient number of file I/O buffer headers are then allocated to allow each to hold 2 pages each. Each buffer maps 8K bytes. If the number of buffer pages is larger than can be mapped by the buffer headers, the number of pages is reduced. The number of buffer headers allocated is stored in the global variable *nbuf*, which may be patched before the system is booted. The system option `options NBUF="1000"` forces the allocation of 1000 buffer headers. Half as many swap I/O buffer headers as file I/O buffers are allocated, but no more than 256.

## System size limitations

As distributed, the sum of the virtual sizes of the core-resident processes is limited to 256M bytes. The size of the text segment of a single process is currently limited to 6M bytes. It may be increased to no greater than the data segment size limit (see below) by redefining MAXTSIZ. This may be done with a configuration file option, e.g. options MAXTSIZ="(10\*1024\*1024)" to set the limit to 10 million bytes. Other per-process limits discussed here may be changed with similar options with names given in parentheses. Soft, user-changeable limits are set to 512K bytes for stack (DFLSSIZ) and 6M bytes for the data segment (DFLDSIZ) by default; these may be increased up to the hard limit with the *setrlimit*(2) system call. The data and stack segment size hard limits are set by a system configuration option to one of 17M, 33M or 64M bytes. One of these sizes is chosen based on the definition of MAXDSIZ; with no option, the limit is 17M bytes; with an option options MAXDSIZ="(32\*1024\*1024)" (or any value between 17M and 33M), the limit is increased to 33M bytes, and values larger than 33M result in a limit of 64M bytes. You must be careful in doing this that you have adequate paging space. As normally configured, the system has 16M or 32M bytes per paging area, depending on disk size. The best way to get more space is to provide multiple, thereby interleaved, paging areas. Increasing the virtual memory limits results in interleaving of swap space in larger sections (from 500K bytes to 1M or 2M bytes).

By default, the virtual memory system allocates enough memory for system page tables mapping user page tables to allow 256 megabytes of simultaneous active virtual memory. That is, the sum of the virtual memory sizes of all (completely- or partially-) resident processes can not exceed this limit. If the limit is exceeded, some process(es) must be swapped out. To increase the amount of resident virtual space possible, you can alter the constant USRPTSIZE (in */sys/vax/vmparam.h*). Each page of system page tables allows 8 megabytes of user virtual memory.

Because the file system block numbers are stored in page table *pg\_blkno* entries, the maximum size of a file system is limited to  $2^{24}$  1024 byte blocks. Thus no file system can be larger than 8 gigabytes.

The number of mountable file systems is set at 20 by the definition of NMOUNT in */sys/h/param.h*. This should be sufficient; if not, the value can be increased up to 255. If you have many disks, it makes sense to make some of them single file systems, and the paging areas don't count in this total.

The limit to the number of files that a process may have open simultaneously is set to 64. This limit is set by the NOFILE definition in */sys/h/param.h*. It may be increased arbitrarily, with the caveat that the user structure expands by 5 bytes for each file, and thus UPAGES (*/sys/vax/machparam.h*) must be increased accordingly.

The amount of physical memory is currently limited to 64 Mb by the size of the index fields in the core-map (*/sys/h/cmaph.h*). The limit may be increased by following instructions in that file to enlarge those fields.

## APPENDIX E. NETWORK CONFIGURATION OPTIONS

The network support in the kernel is self-configuring according to the protocol support options (INET and NS) and the network hardware discovered during autoconfiguration. There are several changes that may be made to customize network behavior due to local restrictions. Within the Internet protocol routines, the following options set in the system configuration file are supported:

### GATEWAY

The machine is to be used as a gateway. This option currently makes only minor changes. First, the size of the network routing hash table is increased. Secondly, machines that have only a single hardware network interface will not forward IP packets; without this option, they will also refrain from sending any error indication to the source of unforwardable packets. Gateways with only a single interface are assumed to have missing or broken interfaces, and will return ICMP unreachable errors to hosts sending them packets to be forwarded.

### TCP\_COMPAT\_42

This option forces the system to limit its initial TCP sequence numbers to positive numbers. Without this option, 4.3BSD systems may have problems with TCP connections to 4.2BSD systems that connect but never transfer data. The problem is a bug in the 4.2BSD TCP; this option should be used during the period of conversion to 4.3BSD.

### IPFORWARDING

Normally, 4.3BSD machines with multiple network interfaces will forward IP packets received that should be resent to another host. If the line "options IPFORWARDING="0"" is in the system configuration file, IP packet forwarding will be disabled.

### IPSENDREDIRECTS

When forwarding IP packets, 4.3BSD IP will note when a packet is forwarded using the same interface on which it arrived. When this is noted, if the source machine is on the directly-attached network, an ICMP redirect is sent to the source host. If the packet was forwarded using a route to a host or to a subnet, a host redirect is sent, otherwise a network redirect is sent. The generation of redirects may be inhibited with the configuration option "options IPSENDREDIRECTS="0"."

### SUBNETSARELOCAL

TCP calculates a maximum segment size to use for each connection, and sends no datagrams larger than that size. This size will be no larger than that supported on the outgoing interface. Furthermore, if the destination is not on the local network, the size will be no larger than 576 bytes. For this test, other subnets of a directly-connected subnetted network are considered to be local unless the line "options SUBNETSARELOCAL="0"" is used in the system configuration file.

### COMPAT\_42

This option, intended as a catchall for 4.2BSD compatibility options, has only a single function thus far. It disables the checking of UDP input packet checksums. As the calculation of UDP packet checksums was incorrect in 4.2BSD, this option allows a 4.3BSD system to receive UDP packets from a 4.2BSD system.

The following options are supported by the Xerox NS protocols:

### NSIP

This option allows NS IDP datagrams to be encapsulated in Internet IP packets for transmission to a collaborating NSIP host. This may be used to pass IDP packets through IP-only link layer networks. See *nsip(4P)* for details.

### THREEWAYSHAKE

The NS Sequenced Packet Protocol does not require a three-way handshake before considering a connection to be in the established state. (A three-way handshake consists of a connection request, an acknowledgement of the request along with a symmetrical opening indication, and then an acknowledgement of the reciprocal opening packet.) This option forces a three-way handshake before data may be transmitted on Sequenced Packet sockets.

# Using ADB to Debug the UNIX<sup>†</sup> Kernel

Samuel J. Leffler and William N. Joy

Computer Systems Research Group  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California, 94720

## ABSTRACT

This document describes the facilities found in the 4.3BSD version of the VAX\* UNIX debugger *adb* which may be used to debug the UNIX kernel. It discusses how standard *adb* commands may be used in examining the kernel and introduces the basics necessary for users to write *adb* command scripts which can augment the standard *adb* command set. The examination techniques described here may be applied both to running systems and the post-mortem dumps automatically created by the *savecore*(8) program after a system crash. The reader is expected to have at least a passing familiarity with the debugger command language.

Revised June 3, 1986

## 1. Introduction

Modifications have been made to the standard VAX UNIX debugger *adb* to simplify examination of post-mortem dumps automatically generated following a system crash. These changes may also be used when examining UNIX in its normal operation. This document serves as an introduction to the use of these facilities, and should not be construed as a description of *how to debug the kernel*.

### 1.1. Invocation

When examining post-mortem dumps of the UNIX kernel the *-k* option should be used, e.g.

```
% adb -k vmunix.? vmcore.?
```

where the appropriate version of the saved operating system image and core dump are supplied in place of "?". This flag causes *adb* to partially simulate the VAX virtual memory hardware when accessing the *core* file. In addition the internal state maintained by the debugger is initialized from data structures maintained by the kernel explicitly for debugging<sup>‡</sup>. A running kernel may be examined in a similar fashion,

```
% adb -k /vmunix /dev/mem
```

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

\*DEC and VAX are trademarks of Digital Equipment Corporation.

<sup>‡</sup> If the *-k* flag is not used when invoking *adb* the user must explicitly calculate virtual addresses. With the *-k* option *adb* interprets page tables to automatically perform virtual to physical address translation.

## 1.2. Establishing Context

During initialization *adb* attempts to establish the context of the “currently active process” by examining the value of the kernel variable *masterpaddr*. This variable contains the virtual address of the process context block of the last process which was set executing by the *Switch* routine. *Masterpaddr* normally provides sufficient information to locate the current stack frame (via the stack pointers found in the context block). By locating the process context block for the process *adb* may then perform virtual to physical address translation using that process’s in-core page tables.

When examining post-mortem dumps locating the most recent stack frame of the last currently active process can be nontrivial. This is due to the different ways in which state may be saved after a nonrecoverable error. Crashes may or may not be “clean” (i.e. the top of the interrupt stack contains a pointer to the process’s kernel mode stack pointer and program counter); an “unclean” crash will occur, for instance, if the interrupt stack overflows. When *adb* is invoked on a post-mortem crash dump it tries to automatically establish the proper stack frame. This is done by first checking the stack pointer normally saved in the restart parameter block at *rpb+1fc* (or *scb-4*). If this value does not point to a valid stack frame, *adb* searches the interrupt stack looking for a valid stack frame. Should this also fail *adb* then searches the kernel stack located in the user structure associated with the last executing process. If *adb* is able to locate a valid stack frame using this procedure the command

**\$c**

will generate a stack trace from the last point at which the kernel was executing on behalf of the user process all the way to the top of the user process’s stack (e.g. to the *main* routine in the user process). Should *adb* be unable to locate a valid stack frame it prints a message and the current state is left undefined. When a stack trace of a particular process (other than that which was currently executing) is desired, an alternate method, described in §2.4, should be used.

Additional information may be obtained from the kernel stack. Discussion of that subject is postponed until command scripts have been introduced; see §2.2.

## 2. Command Scripts

### 2.1. Extending the Formatting Facilities

Once the process context has been established, the complete *adb* command set is available for interpreting data structures. In addition, a number of *adb* scripts have been created to simplify the structured printing of commonly referenced kernel data structures. The scripts normally reside in the directory */usr/lib/adb*, and are invoked with the “\$<” operator. (A later table lists the standard scripts distributed with the system.)

As an example, consider the following listing which contains a dump of a faulty process’s state (our typing is shown emboldened).

```
% adb -k vmunix.175 vmcore.175
sbr 5868 slr 2770
p0br 5a00 p0lr 236 p1br 6600 p1lr fff0
panic: dup biodone
$c
_boot() from _boot+f3
_boot(0,0) from _panic+3a
_panic(800413d0) from _biodone+17
_biodone(800791e8) from _rxpurge+23
_rxpurge(80044754) from _rxstart+5a
_rxstart(80044754) from 80031df8
_rxintr(0) from _Rxintr0+11
_Rxintr0(45b01,3aaf4) from 457f
_Syssize(3aaf4) from 365a
```



[illegible]

[illegible]

```

0      0      0      0
7ffff2d0: minflt      majflt      nswap
0      0      0
7ffff2dc: inblock     oubleck      msgsnd      msgrcv
0      0      0
7ffff2ec: nsignals    nvcsw      nivcsw
0      0      0
7ffff2f8: itimers
0      0      0      0
0      0      0      0
0      0      0      0
7ffff328: XXX
0      0      0
7ffff334: start      acflag
1985 Nov 1 21:27:18 0
7ffff340: pr_base    pr_size    pr_off      scale
0      0      0      0
7ffff350: limits
7fffffff    7fffffff    7fffffff    7fffffff
600000      1000000    80000      1000000
7fffffff    7fffffff    123000     123000
7ffff380: quota      qflags
80074a18    0
7ffff388: nc_off     nc_inum    nc_dev      nc_time
284 2      8 1985 Nov 1 21:27:19
7ffff398: ni_dirp    nameiop    ni_err      ni_pdir      ni_bp
7fffe8a8 41 0 200 800606c4
7ffff3a8: ni_base    ni_count   ni_iovec     ni_iovcnt
0 92 7ffff3a8 1
7ffff3b8: ni_offset  ni_segflg  ni_resid
284 0 0
7ffff3c4: ni_dent.d_inum reclen namlen name
19 72 9 ctm110435^@c^a^a^a
80066de8$<proc
80066de8: link      rlink      next      prev
80044e50    0      80067dec    8004e198

80066df8: addr      upri      pri      cpu      stat      time
802f65d8    0150     0150     0330     03 04
80066e01: nice      slp      cursig    sig
0 0 0 0
80066e08: mask      ignore    catch
0 0 80
80066e14: flag      uid      pgrp      pid      ppid
1008001    2025     11019    11045    11043
80066e20: xstat      ru      poip      szpt      tsize
0 0 0 6 aa
80066e30: dsize      ssize     rssize     maxrss
18c 6 13c 918
80066e40: swrss      swaddr    wchan      textp
0 6d8 0 8006b400
80066e50: p0br      xlink     ticks
802f5a00    0      0
80066e5c: %cpu      ndx      idhash     pptr

```



```

+0.0000000000000000e+00      3ea4      106a      2e
80066e68:  cptr      osptr      ysptr
80067dec      0      0
80066e74:  real itimer
0      0      0      0
80066e84:  quota      0
8006b400$<text
8006b400:  forw      back
1f30      0
daddr
0      0      0      0
0      0      0      0
0      0      2c2      aa

ptdaddr      size      caddr      iptr
80066de8      8005f4a0      74      10001

rssize      swrss      count      ccount      flag      slptim      poip
22 0      0100      031 0      0      0

```

The cause of the crash was a "panic" (see the stack trace) due to an inconsistency recognized inside the *biodone* routine. The majority of the dump was done to illustrate the use of two command scripts used to format kernel data structures. The "u" script, invoked with the command "u\$<u", is a lengthy series of commands which pretty-prints the user structure. Likewise, "proc" and "text" are scripts used to format the obvious data structures. Let's quickly examine the "text" script (the script has been broken into a number of lines for convenience here; in actuality it is a single line of text).

```

./"forw"16t"back"n2Xn\
"daddr"n12Xn\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx++n

```

The first line displays the pointers associated with the doubly linked list used in managing text segments. The second line produces the list of disk block addresses associated with a swapped out text segment. The "n" format forces a new-line character, with 12 hexadecimal integers printed immediately after. Likewise, the remaining two lines of the command format the remainder of the text structure. The expression "16t" causes *adb* to tab to the next column which is a multiple of 16. The last two plus operators are present to round "." to the end of the text structure. This allows the user to reinvoke the format on consecutive text structures without having to be concerned about proper alignment of ".".

The majority of the scripts provided are of this nature. When possible, the formatting scripts print a data structure with a single format to allow subsequent reuse when interrogating arrays of structures. That is, the previous script could have been written

```

./"forw"16t"back"n2Xn
+/"daddr"n12Xn
+/"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn
+/"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx++n

```

but then reuse of the format would have invoked only the last line of the format.

## 2.2. Locating stack frames

It is frequently desirable to locate stack frames in order to examine local and register variables. In particular, frames created by a trap include saved values of all registers and the trap context, and all registers are saved upon a panic as well. Two scripts are provided for tracing stack frames. The first is capable of tracing through multiple frames, printing the information common to each. The second prints all of the information available in the stack frame after a trap. The following example

illustrates their use.

```
% adb -k vmunix.188 vmcore.188
sbr 7068 slr 2770
p0br 5a00 p0lr 74 p1br 5e00 p1lr fff0
panic: Segmentation fault
$c
_boot() from 80029ddb
_boot(0,0) from _panic+3a
_panic(800447a8) from _trap+ac
_trap() from _Xtransflt+1d
_Xtransflt() from _Xsyscall+c
_Xsyscall(7fffe7ac,1b6) from 514
?(7fffe7ac) from 4ac
?() from 196
?(2,7fffe810,7fffe81c) from 3d
?()
1000$s
*(rpb+1fc),4$<frame
7fffe74:  handler      psr      mask
        0          0      2101
        ap         fp      pc
        7fffec0     7fffe9c     80029ddb     _boot+103

7fffe9c:  handler      psr      mask
        0          0      2f00
        ap         fp      pc
        7ffff14     7ffffd0     80012de2     _panic+3a

7ffffd0:  handler      psr      mask
        0          0      2fff
        ap         fp      pc
        7ffff70     7ffff2c     8002a408     _trap+ac

7ffff2c:  handler      psr      mask
        0          0      2fff
        ap         fp      pc
        7ffffe8     7ffffa4     80001031     _Xtransflt+1d

<1$<trapframe
7ffff2c:  handler      psr      mask
        0          0      2fff
        ap         fp      pc
        7ffffe8     7ffffa4     80001031     _Xtransflt+1d
        r0         r1         r2         r3
        0          80046988     80046a00     800728db
        r4         r5         r6         r7
        800728b0     80054158     80063a60     80066ee0
        r8         r9         r10        r11
        80041b80     8          7fffe578     80000000
7ffff70:  nargs      sp      type      code
        0          7fffe560     8          2a50b6ca
        pc         (pc)      ps
        80001651     _Switch+2b     d80008
80001651?
```



```

_Swch+2b: remque *0(r1),r2
80046988/X
_qs:
_qs:      2a50b6ca

```

The example shows a panic due to a segmentation fault. The command "1000\$s" expands the range over which addresses will be displayed symbolically. The back trace indicates that the trap occurred four frames from the end; as the frame pointer is stored at *rpb* 1fc, the command "(rpb+1fc),4\$<frame" prints the last four stack frames; "(rpb+1fc)" is the initial frame pointer, and the count determines the number of frames to print. Having located the stack frame after the trap (the frame with a return PC of Xtransflt+1d), that frame may be displayed again using the script for a trap frame. The previous frame pointer was left in register 1 by the previous script, and thus "<1\$<trapframe" displays the state at the time of the trap. The PC at the time of the fault is shown on the last line from the script, with the faulting address listed as the code in the previous line. The instruction that caused the fault can then be examined. In this example, the instruction was a *remque* that used a displacement addressing mode indirecting through R1. The location to which the register points is the first of the process run queues, and its first element can be seen to be corrupted; its forward pointer, 2a50b6ca, is invalid and is the address that caused the fault.

### 2.3. Traversing Data Structures

The *adb* command language can be used to traverse complex data structures. One data structure, a linked list, occurs quite often in the kernel. By using *adb* variables and the normal expression operators it is a simple matter to construct a script which chains down a list printing each element along the way.

For instance, the queue of processes awaiting timer events, the callout queue, is printed with the following two scripts:

```

callout:
    call todo/"time"16t"arg"16t"func"12+
    *+$<callout.next

callout.next:
    ./Dpp
    *+>1
    ,#<1$<
    <1$<callout.next

```

The first line of the script *callout* starts the traversal at the global symbol *calltodo* and prints a set of headings. It then skips the empty portion of the structure used as the head of the queue. The second line then invokes the script *callout.next* moving "." to the top of the queue ("+" performs the indirection through the *link* entry of the structure at the head of the queue).

*callout.next* prints values for each column, then performs a conditional test on the link to the next entry. This test is performed as follows,

```

*+>1    Place the value of the "link" in the adb variable "<1".
, #<1$<  If the value stored in "<1" is non-zero, then the current input stream (i.e. the script
          callout.next) is terminated. Otherwise, the expression "#<1" will be zero, and the "$<" will
          be ignored. That is, the combination of the logical negation operator "#", the adb variable
          "<1", and the "$<" operator creates a statement of the form,

          if (1link) exit;

```

The remaining line of *callout.next* simply reapplies the script on the next element in the linked list.

A sample *callout* dump is shown below.

```
% adb -k /vmunix /dev/mem
sbr 8001f864 slr d9c
p0br 800efa00 p0lr 8e p1br 7f8efe00 p1lr 1ffff2
$<callout
_calltodo:
_calltodo:  time      arg      func
8004ecfc:  26      0      _dzscan
8004ed0c:  8       0      _upwatch
8004ed1c:  0       0      _ip_timeo
8004ed5c:  0       0      _tcp_timeo
8004ed6c:  0       0      _rkwatch
8004ecfc:  52      0      _dzscan
8004ed2c:  68      _Syssize+70 _tmtimer
8004ed3c:  2920    0      _memeable
```

#### 2.4. Supplying Parameters

If one is clever, a command script may use the address and count portions of an *adb* command as parameters. An example of this is the *setproc* script used to switch to the context of a process with a known process-id;

```
0t99$<setproc
```

The body of *setproc* is

```
.>4
*nproc>l
*proc>f
$<setproc.nxt
```

while *setproc.nxt* is

```
(*(<f+0t52))&0xffff="pid "D
,##(*(<f+0t52)&0xffff)-<4)$<setproc.done
<l-1>l
<f+0t164>f
,##<l$<
$<setproc.nxt
```

The process-id, supplied as the parameter, is stored in the variable "<4", the number of processes is placed in "<l", and the base of the array of process structures in "<f". *setproc.nxt* then performs a linear search through the array until it matches the process-id requested, or until it runs out of process structures to check. The script *setproc.done* simply establishes the context of the process, then exits.

#### 2.5. Standard Scripts

The following table summarizes the command scripts supplied with 4.3BSD; these scripts are found in the directory */usr/lib/adb*.

Standard Command Scripts		
Name	Use	Description
buf	addr\$<buf	format block I/O buffer
callout	\$<callout	print timer queue
clist	addr\$<clist	format character I/O linked list
dino	addr\$<dino	format directory inode
dir	addr\$<dir	format directory entry
dirblk	addr\$<dirblk	scan directory entries



Standard Command Scripts		
Name	Use	Description
dmap	<i>addr\$&lt;dmap</i>	format a disk-map structure
dmccstats	<i>\$&lt;dmccstats</i>	dump statistics for dmcc0
file	<i>addr\$&lt;file</i>	format open file structure
filsys	<i>addr\$&lt;filsys</i>	format in-core super block structure
findinode	<i>inum\$&lt;findinode</i>	find an inode in the in-core inode table
findproc	<i>pid\$&lt;findproc</i>	find process by process id
frame	<i>addr,count\$&lt;frame</i>	trace <i>count</i> stack frames starting at <i>addr</i>
hosts	<i>addr\$&lt;hosts</i>	format IMP host table entries
hosttable	<i>addr\$&lt;hosttable</i>	show all IMP host table entries
ifaddr	<i>addr\$&lt;ifaddr</i>	format a network interface address structure
ifnet	<i>addr\$&lt;ifnet</i>	format network interface structure
ifuba	<i>addr\$&lt;ifuba</i>	format UNIBUS resource structure
imp	<i>addr\$&lt;imp</i>	format an IMP interface state structure
in_ifaddr	<i>addr\$&lt;in_ifaddr</i>	format internet network addresses for an interface
inode	<i>addr\$&lt;inode</i>	format in-core inode structure
inpcb	<i>addr\$&lt;inpcb</i>	format internet protocol control block
iovec	<i>addr\$&lt;iovec</i>	format a list of <i>iov</i> structures
ipreass	<i>addr\$&lt;ipreass</i>	format an ip reassembly queue
mact	<i>addr\$&lt;mact</i>	show "active" list of mbuf's
mba_device	<i>addr\$&lt;mba_device</i>	format an MBA device structure
mba_hd	<i>addr\$&lt;mba_hd</i>	format an MBA queue head
mbstat	<i>\$&lt;mbstat</i>	show mbuf statistics
mbuf	<i>addr\$&lt;mbuf</i>	show "next" list of mbuf's
mbufchain	<i>addr\$&lt;mbufchain</i>	display a chain of mbufs queued at a socket
mbufs	<i>addr\$&lt;mbufs</i>	show a number of mbuf's
mount	<i>addr\$&lt;mount</i>	format mount structure
nameidata	<i>addr\$&lt;nameidata</i>	format a namei parameter block
packetchain	<i>addr\$&lt;packetchain</i>	format a chain of packets
pcb	<i>addr\$&lt;pcb</i>	format process context block
proc	<i>addr\$&lt;proc</i>	format process table entry
protosw	<i>addr\$&lt;protosw</i>	format a protocol switch entry
quota	<i>addr\$&lt;quota</i>	format a disk quota structure
rawcb	<i>addr\$&lt;rawcb</i>	format a raw protocol control block
rtentry	<i>addr\$&lt;rtentry</i>	format a routing table entry
rusage	<i>addr\$&lt;rusage</i>	format a resource usage structure
setproc	<i>pid\$&lt;setproc</i>	switch process context to <i>pid</i>
socket	<i>addr\$&lt;socket</i>	format socket structure
stat	<i>addr\$&lt;stat</i>	format a stat structure
tcpcb	<i>addr\$&lt;tcpcb</i>	format TCP control block
tcpip	<i>addr\$&lt;tcpip</i>	format a TCP/IP packet header
tcpreass	<i>addr\$&lt;tcpreass</i>	show a TCP reassembly queue
text	<i>addr\$&lt;text</i>	format text structure
traceall	<i>\$&lt;traceall</i>	show stack trace for all processes
trapframe	<i>addr\$&lt;trapframe</i>	format a stack frame generated by a trap
tty	<i>addr\$&lt;tty</i>	format tty structure
u	<i>addr\$&lt;u</i>	format user vector, including pcb
ubadev	<i>addr\$&lt;ubadev</i>	format a UBA device structure
ubahd	<i>addr\$&lt;ubahd</i>	format a UNIBUS header structure
unpcb	<i>addr\$&lt;unpcb</i>	format a UNIX domain protocol control block



### 3. Summary

The extensions made to *adb* provide basic support for debugging the UNIX kernel by eliminating the need for a user to carry out virtual to physical address translation and by automatically locating the stack frame after a system crash. A collection of scripts have been written to format the major kernel data structures and aid in switching between process contexts. These facilities have been implemented with only minimal changes to the debugger. While the symbolic debugger *dbx* provides facilities similar to those described here it is not yet a viable alternative to *adb* because *dbx* takes too long to read in the symbol table. As soon as this problem is corrected there will be only limited need for the facilities provided by *adb*.



S  
M  
M

3

## Disc Quotas in a UNIX\* Environment

Robert Elz

Department of Computer Science  
University of Melbourne,  
Parkville,  
Victoria,  
Australia.

### ABSTRACT

In most computing environments, disc space is not infinite. The disc quota system provides a mechanism to control usage of disc space, on an individual basis.

Quotas may be set for each individual user, on any, or all filesystems.

The quota system will warn users when they exceed their allotted limit, but allow some extra space for current work. Repeatedly remaining over quota at logout, will cause a fatal over quota condition eventually.

The quota system is an optional part of VMUNIX that may be included when the system is configured.

### 1. Users' view of disc quotas

To most users, disc quotas will either be of no concern, or a fact of life that cannot be avoided. The *quota*(1) command will provide information on any disc quotas that may have been imposed upon a user.

There are two individual possible quotas that may be imposed, usually if one is, both will be. A limit can be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) he can own.

*Quota* provides information on the quotas that have been set by the system administrators, in each of these areas, and current usage.

There are four numbers for each limit, the current usage, soft limit (quota), hard limit, and number of remaining login warnings. The soft limit is the number of 1K blocks (or files) that the user is expected to remain below. Each time the user's usage goes past this limit, he will be warned. The hard limit cannot be exceeded. If a user's usage reaches this number, further requests for space (or attempts to create a file) will fail with an EDQUOT error, and the first time this occurs, a message will be written to the user's terminal. Only one message will be output, until space occupied is reduced below the limit, and reaches it again, in order to avoid continual noise from those programs that ignore write errors.

Whenever a user logs in with a usage greater than his soft limit, he will be warned, and his login warning count decremented. When he logs in under quota, the counter is reset to its maximum value (which is a system configuration parameter, that is typically 3). If the warning count should ever reach zero (caused by three successive logins over quota), the particular limit that has been exceeded will be treated as if the hard limit has been reached, and no

\* UNIX is a trademark of Bell Laboratories.

more resources will be allocated to the user. The only way to reset this condition is to reduce usage below quota, then log in again.

### 1.1. Surviving when quota limit is reached

In most cases, the only way to recover from over quota conditions, is to abort whatever activity was in progress on the filesystem that has reached its limit, remove sufficient files to bring the limit back below quota, and retry the failed program.

However, if you are in the editor and a write fails because of an over quota situation, that is not a suitable course of action, as it is most likely that initially attempting to write the file will have truncated its previous contents, so should the editor be aborted without correctly writing the file not only will the recent changes be lost, but possibly much, or even all, of the data that previously existed.

There are several possible safe exits for a user caught in this situation. He may use the editor ! shell escape command to examine his file space, and remove surplus files. Alternatively, using *csh*, he may suspend the editor, remove some files, then resume it. A third possibility, is to write the file to some other filesystem (perhaps to a file on /tmp) where the user's quota has not been exceeded. Then after rectifying the quota situation, the file can be moved back to the filesystem it belongs on.

## 2. Administering the quota system

To set up and establish the disc quota system, there are several steps necessary to be performed by the system administrator.

First, the system must be configured to include the disc quota sub-system. This is done by including the line:

```
options QUOTA
```

in the system configuration file, then running *config(8)* followed by a system configuration\*.

Second, a decision as to what filesystems need to have quotas applied needs to be made. Usually, only filesystems that house users' home directories, or other user files, will need to be subjected to the quota system, though it may also prove useful to also include /usr. If possible, /tmp should usually be free of quotas.

Having decided on which filesystems quotas need to be set upon, the administrator should then allocate the available space amongst the competing needs. How this should be done is (way) beyond the scope of this document.

Then, the *edquota(8)* command can be used to actually set the limits desired upon each user. Where a number of users are to be given the same quotas (a common occurrence) the *-p* switch to *edquota* will allow this to be easily accomplished.

Once the quotas are set, ready to operate, the system must be informed to enforce quotas on the desired filesystems. This is accomplished with the *quotaon(8)* command. *Quotaon* will either enable quotas for a particular filesystem, or with the *-a* switch, will enable quotas for each filesystem indicated in */etc/fstab* as using quotas. See *fstab(5)* for details. Most sites using the quota system, will include the line

```
/etc/quotaon -a
```

in */etc/rc.local*.

Should quotas need to be disabled, the *quotaoff(8)* command will do that, however, should the filesystem be about to be dismounted, the *umount(8)* command will disable quotas immediately before the filesystem is unmounted. This is actually an effect of the *umount(2)* system call, and it guarantees that the quota system will not be disabled if the *umount* would

\* See also the document "Building 4.2BSD UNIX Systems with Config".

fail because the filesystem is not idle.

Periodically (certainly after each reboot, and when quotas are first enabled for a filesystem), the records retained in the quota file should be checked for consistency with the actual number of blocks and files allocated to the user. The *quotachk*(8) command can be used to accomplish this. It is not necessary to dismount the filesystem, or disable the quota system to run this command, though on active filesystems inaccurate results may occur. This does no real harm in most cases, another run of *quotachk* when the filesystem is idle will certainly correct any inaccuracy.

The super-user may use the *quota*(1) command to examine the usage and quotas of any user, and the *repquota*(8) command may be used to check the usages and limits for all users on a filesystem.

### 3. Some implementation detail.

Disc quota usage and information is stored in a file on the filesystem that the quotas are to be applied to. Conventionally, this file is *quotas* in the root of the filesystem. While this name is not known to the system in any way, several of the user level utilities "know" it, and choosing any other name would not be wise.

The data in the file comprises an array of structures, indexed by uid, one structure for each user on the system (whether the user has a quota on this filesystem or not). If the uid space is sparse, then the file may have holes in it, which would be lost by copying, so it is best to avoid this.

The system is informed of the existence of the quota file by the *setquota*(2) system call. It then reads the quota entries for each user currently active, then for any files open owned by users who are not currently active. Each subsequent open of a file on the filesystem, will be accompanied by a pairing with its quota information. In most cases this information will be retained in core, either because the user who owns the file is running some process, because other files are open owned by the same user, or because some file (perhaps this one) was recently accessed. In memory, the quota information is kept hashed by user-id and filesystem, and retained in an LRU chain so recently released data can be easily reclaimed. Information about those users whose last process has recently terminated is also retained in this way.

Each time a block is accessed or released, and each time an inode is allocated or freed, the quota system gets told about it, and in the case of allocations, gets the opportunity to object.

Measurements have shown that the quota code uses a very small percentage of the system cpu time consumed in writing a new block to disc.

### 4. Acknowledgments

The current disc quota system is loosely based upon a very early scheme implemented at the University of New South Wales, and Sydney University in the mid 70's. That system implemented a single combined limit for both files and blocks on all filesystems.

A later system was implemented at the University of Melbourne by the author, but was not kept highly accurately, eg: chown's (etc) did not affect quotas, nor did i/o to a file other than one owned by the instigator.

The current system has been running (with only minor modifications) since January 82 at Melbourne. It is actually just a small part of a much broader resource control scheme, which is capable of controlling almost anything that is usually uncontrolled in unix. The rest of this is, as yet, still in a state where it is far too subject to change to be considered for distribution.

For the 4.2BSD release, much work has been done to clean up and sanely incorporate the quota code by Sam Leffler and Kirk McKusick at The University of California at Berkeley.



S  
M  
M

4

## Fsck – The UNIX<sup>†</sup> File System Check Program

*Marshall Kirk McKusick*

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

*T. J. Kowalski*

Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

This document reflects the use of *fsck* with the 4.2BSD and 4.3BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, *fsck* is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

Revised July 16, 1985

<sup>†</sup>UNIX is a trademark of Bell Laboratories.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.



**TABLE OF CONTENTS****1. Introduction****2. Overview of the file system**

- 2.1. Superblock
- 2.2. Summary Information
- 2.3. Cylinder groups
- 2.4. Fragments
- 2.5. Updates to the file system

**3. Fixing corrupted file systems**

- 3.1. Detecting and correcting corruption
- 3.2. Super block checking
- 3.3. Free block checking
- 3.4. Checking the inode state
- 3.5. Inode links
- 3.6. Inode data size
- 3.7. Checking the data associated with an inode
- 3.8. File system connectivity

**Acknowledgements****References****4. Appendix A**

- 4.1. Conventions
- 4.2. Initialization
- 4.3. Phase 1 - Check Blocks and Sizes
- 4.4. Phase 1b - Rescan for more Dups
- 4.5. Phase 2 - Check Pathnames
- 4.6. Phase 3 - Check Connectivity
- 4.7. Phase 4 - Check Reference Counts
- 4.8. Phase 5 - Check Cyl groups
- 4.9. Cleanup



## 1. Introduction

This document reflects the use of *fsck* with the 4.2BSD and 4.3BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

## 2. Overview of the file system

The file system is discussed in detail in [Mckusick84]; this section gives a brief overview.

### 2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks (the triple indirect block is never needed in practice).

In order to create files with up to 2<sup>32</sup> bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

†The actual number may vary from system to system, but is usually in the range 5-13.



## 2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

## 2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the  $i+1$ st cylinder group is about one track further from the beginning of the cylinder group than it was for the  $i$ th cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

## 2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

## 2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk

update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync(2)* is done (at 30 second intervals) by */etc/update(8)*, or by manual operator intervention with the *sync(8)* command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

### 3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, *e.g.*, not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

#### 3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to choose a corrective action.

A quiescent<sup>‡</sup> file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system must be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data

<sup>‡</sup> *I.e.*, unmounted and not being written on.



blocks containing directory entries.

### 3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

### 3.3. Free block checking

*Fsck* checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

### 3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

### 3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the

descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

*Fsck* compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

*Fsck* checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

### 3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

### 3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware

failure.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." must be the first entry in the directory data block. The inode number for "." must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for ".." must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values. If there are multiple hard links to a directory, the first one encountered is considered the real parent to which "." should point; *fsck* recommends deletion for the subsequently discovered names.

### 3.8. File system connectivity

*Fsck* checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

### Acknowledgements

I thank Bill Joy, Sam Leffler, Robert Elz and Dennis Ritchie for their suggestions and help in implementing the new file system. Thanks also to Robert Henry for his editorial input to get this document together. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. (Kirk McKusick, July 1983)

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS. (T. Kowalski, July 1979)

### References

- [Dolotta78] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1*, January 1978.
- [Joy83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. 4.2BSD System Manual, *University of California at Berkeley, Computer Systems Research Group Technical Report #4*, 1982.
- [McKusick84] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX, *ACM Transactions on Computer Systems* 2, 3. pp. 181-197, August 1984.
- [Ritchie78] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.
- [Thompson78] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

## 4. Appendix A – Fsk Error Conditions

### 4.1. Conventions

*Fsk* is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the map of free blocks, (possibly rebuilding it), and performs some cleanup.

Normally *fsck* is run non-interactively to *preen* the file systems after an unclean halt. While *preen*'ing a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. When *preen*'ing most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

### 4.2. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All the initialization errors are fatal when the file system is being *preen*'ed.

#### *C* option?

*C* is not a legal option to *fsck*; legal options are *-b*, *-y*, *-n*, and *-p*. *Fsk* terminates on this error condition. See the *fsck*(8) manual entry for further detail.

cannot alloc NNN bytes for blockmap

cannot alloc NNN bytes for freemap

cannot alloc NNN bytes for statemap

cannot alloc NNN bytes for Incntp

*Fsk*'s request for memory for its virtual memory tables failed. This should never happen. *Fsk* terminates on this error condition. See a guru.

#### Can't open checklist file: *F*

The file system checklist file *F* (usually */etc/fstab*) can not be opened for reading. *Fsk* terminates on this error condition. Check access modes of *F*.

#### Can't stat root

*Fsk*'s request for statistics about the root directory */* failed. This should never happen. *Fsk* terminates on this error condition. See a guru.

#### Can't stat *F*

##### Can't make sense out of name *F*

*Fsk*'s request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.



**Can't open *F***

*Fsck*'s request attempt to open the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

***F*: (NO WRITE)**

Either the *-n* flag was specified or *fsck*'s attempt to open the file system *F* for writing failed. When running manually, all the diagnostics are printed out, but no modifications are attempted to fix them.

**file is not a block or character device; OK**

You have given *fsck* a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES ignore this error condition.

NO ignore this file system and continues checking the next file system given.

**UNDEFINED OPTIMIZATION IN SUPERBLOCK (SET TO DEFAULT)**

The superblock optimization parameter is neither *OPT\_TIME* nor *OPT\_SPACE*.

Possible responses to the SET TO DEFAULT prompt are:

YES The superblock is set to request optimization to minimize running time of the system. (If optimization to minimize disk space utilization is desired, it can be set using *tune2fs(8)*.)

NO ignore this error condition.

**IMPOSSIBLE MINFREE=*D* IN SUPERBLOCK (SET TO DEFAULT)**

The superblock minimum space percentage is greater than 99% or less than 0%.

Possible responses to the SET TO DEFAULT prompt are:

YES The minfree parameter is set to 10%. (If some other percentage is desired, it can be set using *tune2fs(8)*.)

NO ignore this error condition.

One of the following messages will appear:

**MAGIC NUMBER WRONG**

**NCG OUT OF RANGE**

**CPG OUT OF RANGE**

**NCYL DOES NOT JIVE WITH NCG\*CPG**

**SIZE PREPOSTEROUSLY LARGE**

**TRASHED VALUES IN SUPER BLOCK**

and will be followed by the message:

***F*: BAD SUPER BLOCK: *B***

**USE -b OPTION TO FSCK TO SPECIFY LOCATION OF AN ALTERNATE**

**SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE *fsck(8)*.**

The super block has been corrupted. An alternative super block must be selected from among those listed by *newfs(8)* when the file system was created. For file systems with a blocksize less than 32K, specifying *-b 32* is a good first choice.

**INTERNAL INCONSISTENCY: *M***

*Fsck*'s has had an internal panic, whose message is specified as *M*. This should never happen. See a guru.



**CAN NOT SEEK: BLK *B* (CONTINUE)**

*Fsck*'s request for moving to a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

**YES** attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

**NO** terminate the program.

**CAN NOT READ: BLK *B* (CONTINUE)**

*Fsck*'s request for reading a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

**YES** attempt to continue to run the file system check. It will retry the read and print out the message:

**THE FOLLOWING SECTORS COULD NOT BE READ: *N***

where *N* indicates the sectors that could not be read. If *fsck* ever tries to write back one of the blocks on which the read failed it will print the message:

**WRITING ZERO'ED BLOCK *N* TO DISK**

where *N* indicates the sector that was written with zero's. If the disk is experiencing hardware problems, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

**NO** terminate the program.

**CAN NOT WRITE: BLK *B* (CONTINUE)**

*Fsck*'s request for writing a specified block number *B* in the file system failed. The disk is write-protected; check the write protect lock on the drive. If that is not the problem, see a guru.

Possible responses to the CONTINUE prompt are:

**YES** attempt to continue to run the file system check. The write operation will be retried with the failed blocks indicated by the message:

**THE FOLLOWING SECTORS COULD NOT BE WRITTEN: *N***

where *N* indicates the sectors that could not be written. If the disk is experiencing hardware problems, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

**NO** terminate the program.

**bad inode number *DDD* to ginode**

An internal error has attempted to read non-existent inode *DDD*. This error causes *fsck* to exit. See a guru.

**4.3. Phase 1 - Check Blocks and Sizes**

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format. All errors in this phase except **INCORRECT BLOCK COUNT** and **PARTIALLY TRUNCATED**

**INODE** are fatal if the file system is being preen'ed.

#### UNKNOWN FILE TYPE I=I (CLEAR)

The mode word of the inode *I* indicates that the inode is not a special block inode, special character inode, socket inode, regular inode, symbolic link, or directory inode.

Possible responses to the CLEAR prompt are:

**YES** de-allocate inode *I* by zeroing its contents. This will always invoke the **UNALLOCATED** error condition in Phase 2 for each directory entry pointing to this inode.

**NO** ignore this error condition.

#### PARTIALLY TRUNCATED INODE I=I (SALVAGE)

*Fsck* has found inode *I* whose size is shorter than the number of blocks allocated to it. This condition should only occur if the system crashes while in the midst of truncating a file. When preen'ing the file system, *fsck* completes the truncation to the specified size.

Possible responses to SALVAGE are:

**YES** complete the truncation to the size specified in the inode.

**NO** ignore this error condition.

#### LINK COUNT TABLE OVERFLOW (CONTINUE)

An internal table for *fsck* containing allocated inodes with a link count of zero cannot allocate more memory. Increase the virtual memory for *fsck*.

Possible responses to the CONTINUE prompt are:

**YES** continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

**NO** terminate the program.

#### B BAD I=I

Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the **EXCESSIVE BAD BLKS** error condition in Phase 1 (see next paragraph) if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the **BAD/DUP** error condition in Phase 2 and Phase 4.

#### EXCESSIVE BAD BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode *I*.

Possible responses to the CONTINUE prompt are:

**YES** ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

**NO** terminate the program.

#### BAD STATE DDD TO BLKERR

An internal error has scrambled *fsck*'s state map to have the impossible value *DDD*. *Fsck* exits immediately. See a guru.

**B DUP I=I**

Inode *I* contains block number *B* that is already claimed by another inode. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the **BAD/DUP** error condition in Phase 2 and Phase 4.

**EXCESSIVE DUP BLKS I=I (CONTINUE)**

There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to the **CONTINUE** prompt are:

**YES** ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

**NO** terminate the program.

**DUP TABLE OVERFLOW (CONTINUE)**

An internal table in *fsck* containing duplicate block numbers cannot allocate any more space. Increase the amount of virtual memory available to *fsck*.

Possible responses to the **CONTINUE** prompt are:

**YES** continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

**NO** terminate the program.

**PARTIALLY ALLOCATED INODE I=I (CLEAR)**

Inode *I* is neither allocated nor unallocated.

Possible responses to the **CLEAR** prompt are:

**YES** de-allocate inode *I* by zeroing its contents.

**NO** ignore this error condition.

**INCORRECT BLOCK COUNT I=I (X should be Y) (CORRECT)**

The block count for inode *I* is *X* blocks, but should be *Y* blocks. When preening the count is corrected.

Possible responses to the **CORRECT** prompt are:

**YES** replace the block count of inode *I* with *Y*.

**NO** ignore this error condition.

**4.4. Phase 1B: Rescan for More Dups**

When a duplicate block is found in the file system, the file system is rescanned to find the inode that previously claimed that block. This section lists the error condition when the duplicate block is found.

**B DUP I=I**

Inode *I* contains block number *B* that is already claimed by another inode. This error condition will always invoke the **BAD/DUP** error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the **DUP** error condition in Phase 1.



#### 4.5. Phase 2 – Check Pathnames

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes, and directory integrity checks. All errors in this phase are fatal if the file system is being preen'ed, except for directories not being a multiple of the blocks size and extraneous hard links.

##### ROOT INODE UNALLOCATED (ALLOCATE)

The root inode (usually inode number 2) has no allocate mode bits. This should never happen.

Possible responses to the ALLOCATE prompt are:

YES allocate inode 2 as the root inode. The files and directories usually found in the root will be recovered in Phase 3 and put into *lost+found*. If the attempt to allocate the root fails, *fsck* will exit with the message:

**CANNOT ALLOCATE ROOT INODE.**

NO *fsck* will exit.

##### ROOT INODE NOT DIRECTORY (REALLOCATE)

The root inode (usually inode number 2) is not directory inode type.

Possible responses to the REALLOCATE prompt are:

YES clear the existing contents of the root inode and reallocate it. The files and directories usually found in the root will be recovered in Phase 3 and put into *lost+found*. If the attempt to allocate the root fails, *fsck* will exit with the message:

**CANNOT ALLOCATE ROOT INODE.**

NO *fsck* will then prompt with **FIX**

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, many error conditions will be produced.

NO terminate the program.

##### DUPS/BAD IN ROOT INODE (REALLOCATE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to the REALLOCATE prompt are:

YES clear the existing contents of the root inode and reallocate it. The files and directories usually found in the root will be recovered in Phase 3 and put into *lost+found*. If the attempt to allocate the root fails, *fsck* will exit with the message:

**CANNOT ALLOCATE ROOT INODE.**

NO *fsck* will then prompt with **CONTINUE**.

Possible responses to the CONTINUE prompt are:

YES ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in many other error conditions.

NO terminate the program.

##### NAME TOO LONG F

An excessively long path name has been found. This usually indicates loops in the file system

name space. This can occur if the super user has made circular links to directories. The offending links must be removed (by a guru).

#### **I OUT OF RANGE I=*I* NAME=*F* (REMOVE)**

A directory entry *F* has an inode number *I* that is greater than the end of the inode list.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

#### **UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* type=*F* (REMOVE)**

A directory or file entry *F* points to an unallocated inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

#### **DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* type=*F* (REMOVE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory or file entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

#### **ZERO LENGTH DIRECTORY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)**

A directory entry *F* has a size *S* that is zero. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed; this will always invoke the BAD/DUP error condition in Phase 4.

NO ignore this error condition.

#### **DIRECTORY TOO SHORT I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)**

A directory *F* has been found whose size *S* is less than the minimum size directory. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the FIX prompt are:

YES increase the size of the directory to the minimum directory size.

NO ignore this directory.

#### **DIRECTORY *F* LENGTH *S* NOT MULTIPLE OF *B* (ADJUST)**

A directory *F* has been found with size *S* that is not a multiple of the directory blocksize *B*.

Possible responses to the ADJUST prompt are:

YES the length is rounded up to the appropriate block size. This error can occur on 4.2BSD file systems. Thus when preening the file system only a warning is printed and the directory is adjusted.



NO ignore the error condition.

**DIRECTORY CORRUPTED I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (SALVAGE)**

A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES throw away all entries up to the next directory boundary (usually 512-byte) boundary. This drastic action can throw away up to 42 entries, and should be taken only after other recovery efforts have failed.

NO skip up to the next directory boundary and resume reading, but do not modify the directory.

**BAD INODE NUMBER FOR '.' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found whose inode number for '.' does not equal *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '.' to be equal to *I*.

NO leave the inode number for '.' unchanged.

**MISSING '.' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES build an entry for '.' with inode number equal to *I*.

NO leave the directory unchanged.

**MISSING '.' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F**

**CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS F**

A directory *I* has been found whose first entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

**MISSING '.' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F**

**CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'**

A directory *I* has been found whose first entry is not '.'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

**EXTRA '.' ENTRY I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '.'.

NO leave the directory unchanged.

**BAD INODE NUMBER FOR '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found whose inode number for '..' does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '..' to be equal to the parent of *I* (".." in the root inode points to itself).

NO leave the inode number for '..' unchanged.

**MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES build an entry for '..' with inode number equal to the parent of *I* (".." in the root inode points to itself).

NO leave the directory unchanged.

**MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F**

**CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS F**

A directory *I* has been found whose second entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

**MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F**

**CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'**

A directory *I* has been found whose second entry is not '..'. *Fsck* cannot resolve this problem. The file system should be mounted and the second entry in the directory moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

**EXTRA '..' ENTRY I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)**

A directory *I* has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '..'.

NO leave the directory unchanged.

**N IS AN EXTRANEIOUS HARD LINK TO A DIRECTORY D (REMOVE)**

*Fsck* has found a hard link, *N*, to a directory, *D*. When preening the extraneous links are ignored.

Possible responses to the REMOVE prompt are:

YES delete the extraneous entry, *N*.

NO ignore the error condition.

**BAD INODE S TO DESCEND**

An internal error has caused an impossible state *S* to be passed to the routine that descends the file system directory structure. *Fsck* exits. See a guru.

**BAD RETURN STATE S FROM DESCEND**

An internal error has caused an impossible state *S* to be returned from the routine that descends the file system directory structure. *Fsck* exits. See a guru.

**BAD STATE S FOR ROOT INODE**

An internal error has caused an impossible state *S* to be assigned to the root inode. *Fsck* exits. See a guru.



#### 4.6. Phase 3 - Check Connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

##### UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)

The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing, the directory is reconnected if its size is non-zero, otherwise it is cleared.

Possible responses to the RECONNECT prompt are:

**YES** reconnect directory inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

**NO** ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

##### NO *lost+found* DIRECTORY (CREATE)

There is no *lost+found* directory in the root directory of the file system; When preen'ing *fsck* tries to create a *lost+found* directory.

Possible responses to the CREATE prompt are:

**YES** create a *lost+found* directory in the root of the file system. This may raise the message:

##### NO SPACE LEFT IN / (EXPAND)

See below for the possible responses. Inability to create a *lost+found* directory generates the message:

##### SORRY. CANNOT CREATE *lost+found* DIRECTORY

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**NO** abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

##### *lost+found* IS NOT A DIRECTORY (REALLOCATE)

The entry for *lost+found* is not a directory.

Possible responses to the REALLOCATE prompt are:

**YES** allocate a directory inode, and change *lost+found* to reference it. The previous inode reference by the *lost+found* name is not cleared. Thus it will either be reclaimed as an UNREF'ed inode or have its link count ADJUST'ed later in this Phase. Inability to create a *lost+found* directory generates the message:

##### SORRY. CANNOT CREATE *lost+found* DIRECTORY

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**NO** abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

##### NO SPACE LEFT IN /*lost+found* (EXPAND)

There is no space to add another entry to the *lost+found* directory in the root directory of the file system. When preen'ing the *lost+found* directory is expanded.

Possible responses to the EXPAND prompt are:

**YES** the *lost+found* directory is expanded to make room for the new entry. If the attempted expansion fails *fsck* prints the message:



**SORRY. NO SPACE IN *lost+found* DIRECTORY**

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found*. This error is fatal if the file system is being preen'ed.

NO abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**DIR *I*=*I1* CONNECTED. PARENT WAS *I*=*I2***

This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

**DIRECTORY *F* LENGTH *S* NOT MULTIPLE OF *B* (ADJUST)**

A directory *F* has been found with size *S* that is not a multiple of the directory blocksize *B* (this can reoccur in Phase 3 if it is not adjusted in Phase 2).

Possible responses to the ADJUST prompt are:

YES the length is rounded up to the appropriate block size. This error can occur on 4.2BSD file systems. Thus when preen'ing the file system only a warning is printed and the directory is adjusted.

NO ignore the error condition.

**BAD INODE *S* TO DESCEND**

An internal error has caused an impossible state *S* to be passed to the routine that descends the file system directory structure. *Fsck* exits. See a guru.

**4.7. Phase 4 - Check Reference Counts**

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, symbolic links, or special files, unreferenced files, symbolic links, and directories, and bad or duplicate blocks in files, symbolic links, and directories. All errors in this phase are correctable if the file system is being preen'ed except running out of space in the *lost+found* directory.

**UNREF FILE *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)**

Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing the file is cleared if either its size or its link count is zero, otherwise it is reconnected.

Possible responses to the RECONNECT prompt are:

YES reconnect inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.

NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

**(CLEAR)**

The inode mentioned in the immediately previous error condition can not be reconnected. This cannot occur if the file system is being preen'ed, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:



**YES** de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

**NO** ignore this error condition.

**NO *lost+found* DIRECTORY (CREATE)**

There is no *lost+found* directory in the root directory of the file system; When preening *fsck* tries to create a *lost+found* directory.

Possible responses to the CREATE prompt are:

**YES** create a *lost+found* directory in the root of the file system. This may raise the message:  
**NO SPACE LEFT IN / (EXPAND)**

See below for the possible responses. Inability to create a *lost+found* directory generates the message:

**SORRY. CANNOT CREATE *lost+found* DIRECTORY**

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**NO** abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

***lost+found* IS NOT A DIRECTORY (REALLOCATE)**

The entry for *lost+found* is not a directory.

Possible responses to the REALLOCATE prompt are:

**YES** allocate a directory inode, and change *lost+found* to reference it. The previous inode reference by the *lost+found* name is not cleared. Thus it will either be reclaimed as an UNREF'ed inode or have its link count ADJUST'ed later in this Phase. Inability to create a *lost+found* directory generates the message:

**SORRY. CANNOT CREATE *lost+found* DIRECTORY**

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**NO** abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**NO SPACE LEFT IN /*lost+found* (EXPAND)**

There is no space to add another entry to the *lost+found* directory in the root directory of the file system. When preening the *lost+found* directory is expanded.

Possible responses to the EXPAND prompt are:

**YES** the *lost+found* directory is expanded to make room for the new entry. If the attempted expansion fails *fsck* prints the message:

**SORRY. NO SPACE IN *lost+found* DIRECTORY**

and aborts the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found*. This error is fatal if the file system is being preened.

**NO** abort the attempt to linkup the lost inode. This will always invoke the UNREF error condition in Phase 4.

**LINK COUNT *type* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)**

The link count for inode *I*, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preening the link count is adjusted unless the number of references is increasing, a condition that should never occur unless precipitated by a hardware failure. When the number of references is increasing under preen mode, *fsck* exits with the

message:

**LINK COUNT INCREASING**

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode *I* with *Y*.

NO ignore this error condition.

**UNREF type *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)**

Inode *I*, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preening, this is a file that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

**BAD/DUP type *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preened, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

**4.8. Phase 5 - Check Cyl groups**

This phase concerns itself with the free-block and used-inode maps. This section lists error conditions resulting from allocated blocks in the free-block maps, free blocks missing from free-block maps, and the total free-block count incorrect. It also lists error conditions resulting from free inodes in the used-inode maps, allocated inodes missing from used-inode maps, and the total used-inode count incorrect.

**CG C: BAD MAGIC NUMBER**

The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed. This error is fatal if the file system is being preened.

**BLK(S) MISSING IN BIT MAPS (SALVAGE)**

A cylinder group block map is missing some free blocks. During preening the maps are reconstructed.

Possible responses to the SALVAGE prompt are:

YES reconstruct the free block map.

NO ignore this error condition.

**SUMMARY INFORMATION BAD (SALVAGE)**

The summary information was found to be incorrect. When preening, the summary information is recomputed.

Possible responses to the SALVAGE prompt are:

YES reconstruct the summary information.

NO ignore this error condition.



**FREE BLK COUNT(S) WRONG IN SUPERBLOCK (SALVAGE)**

The superblock free block information was found to be incorrect. When preen'ing, the superblock free block information is recomputed.

Possible responses to the SALVAGE prompt are:

YES reconstruct the superblock free block information.

NO ignore this error condition.

**4.9. Cleanup**

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

*V* files, *W* used, *X* free (*Y* frags, *Z* blocks)

This is an advisory message indicating that the file system checked contained *V* files using *W* fragment sized blocks leaving *X* fragment sized blocks free in the file system. The numbers in parenthesis breaks the free count down into *Y* free fragments and *Z* free full sized blocks.

**\*\*\*\*\* REBOOT UNIX \*\*\*\*\***

This is an advisory message indicating that the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps. When preen'ing, *fsck* will exit with a code of 4. The standard auto-reboot script distributed with 4.3BSD interprets an exit code of 4 by issuing a reboot system call.

**\*\*\*\*\* FILE SYSTEM WAS MODIFIED \*\*\*\*\***

This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

## 4.3BSD Line Printer Spooler Manual

*Ralph Campbell*

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

### ABSTRACT

This document describes the structure and installation procedure for the line printer spooling system developed for the 4.3BSD version of the UNIX\* operating system.

Revised May 14, 1986

#### 1. Overview

The line printer system supports:

- multiple printers,
- multiple spooling queues,
- both local and remote printers, and
- printers attached via serial lines that require line initialization such as the baud rate.

Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

The line printer system consists mainly of the following files and commands:

/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, does all the real work
/usr/ucb/lpr	program to enter a job in a printer queue
/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/etc/lpc	program to administer printers and spooling queues
/dev/printer	socket on which lpd listens

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page entry *printcap(5)* provides the authoritative definition of the format of this data base, as well as specifying default values for important items such as the directory in which spooling is performed. This document introduces some of the information that may be placed *printcap*.

---

\* UNIX is a trademark of Bell Laboratories.



## 2. Commands

### 2.1. *lpd* – line printer daemon

The program *lpd*(8), usually invoked at boot time from the */etc/rc* file, acts as a master server for coordinating and controlling the spooling queues configured in the *printcap* file. When *lpd* is started it makes a single pass through the *printcap* database restarting any printers that have jobs. In normal operation *lpd* listens for service requests on multiple sockets, one in the UNIX domain (named *"/dev/printer"*) for local requests, and one in the Internet domain (under the *"printer"* service specification) for requests for printer access from off machine; see *socket*(2) and *services*(5) for more information on sockets and service specifications, respectively. *Lpd* spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with *lpd* using a simple transaction oriented protocol. Authentication of remote clients is done based on the *"privilege port"* scheme employed by *rshd*(8C) and *rcmd*(3X). The following table shows the requests understood by *lpd*. In each request the first byte indicates the *"meaning"* of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

Request	Interpretation
^Aprinter\n	check the queue for jobs and print any found
^Bprinter\n	receive and queue a job from another machine
^Cprinter [users ...] [jobs ...]\n	return short list of current queue state
^Dprinter [users ...] [jobs ...]\n	return long list of current queue state
^Eprinter person [users ...] [jobs ...]\n	remove jobs from a queue

The *lpr*(1) command is used by users to enter a print job in a local queue and to notify the local *lpd* that there are new jobs in the spooling area. *Lpd* either schedules the job to be printed locally, or if printing remotely, attempts to forward the job to the appropriate machine. If the printer cannot be opened or the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

### 2.2. *lpq* – show line printer queue

The *lpq*(1) program works recursively backwards displaying the queue of the machine with the printer and then the queue(s) of the machine(s) that lead to it. *Lpq* has two forms of output: in the default, short, format it gives a single line of output per queued job; in the long format it shows the list of files, and their sizes, that comprise a job.

### 2.3. *lprm* – remove jobs from a queue

The *lprm*(1) command deletes jobs from a spooling queue. If necessary, *lprm* will first kill off a running daemon that is servicing the queue and restart it after the required files are removed. When removing jobs destined for a remote printer, *lprm* acts similarly to *lpq* except it first checks locally for jobs to remove and then tries to remove files in queues off-machine.

### 2.4. *lpc* – line printer control program

The *lpc*(8) program is used by the system administrator to control the operation of the line printer system. For each line printer configured in */etc/printcap*, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer daemons.

### 3. Access control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The strategy used to maintain protected spooling areas is as follows:

- The spooling area is writable only by a *daemon* user and *daemon* group.
- The *lpr* program runs set-user-id to *root* and set-group-id to group *daemon*. The *root* access permits reading any file required. Accessibility is verified with an *access(2)* call. The group ID is used in setting up proper ownership of files in the spooling area for *lprm*.
- Control files in a spooling area are made with *daemon* ownership and group ownership *daemon*. Their mode is 0660. This insures control files are not modified by a user and that no user can remove files except through *lprm*.
- The spooling programs, *lpd*, *lpq*, and *lprm* run set-user-id to *root* and set-group-id to group *daemon* to access spool files and printers.
- The printer server, *lpd*, uses the same verification procedures as *rshd(8C)* in authenticating remote clients. The host on which a client resides must be present in the file */etc/hosts.equiv* or */etc/hosts.lpd* and the request message must come from a reserved port number.

In practice, none of *lpd*, *lpq*, or *lprm* would have to run as user *root* if remote spooling were not supported. In previous incarnations of the printer system *lpd* ran set-user-id to *daemon*, set-group-id to group *spooling*, and *lpq* and *lprm* ran set-group-id to group *spooling*.

### 4. Setting up

The 4.3BSD release comes with the necessary programs installed and with the default line printer queue created. If the system must be modified, the makefile in the directory */usr/src/usr.lib/lpr* should be used in recompiling and reinstalling the necessary programs.

The real work in setting up is to create the *printcap* file and any printer filters for printers not supported in the distribution system.

#### 4.1. Creating a printcap file

The *printcap* database contains one or more entries per printer. A printer should have a separate spooling directory; otherwise, jobs will be printed on different printers depending on which printer daemon starts first. This section describes how to create entries for printers that do not conform to the default printer description (an LP-11 style interface to a standard, band printer).

##### 4.1.1. Printers on serial lines

When a printer is connected via a serial communication line it must have the proper baud rate and terminal modes set. The following example is for a DecWriter III printer connected locally via a 1200 baud serial line.

```
lp|LA-180 DecWriter III:\
:lp=/dev/lp:br#1200:fs#06320:\
:tr=\f:of=/usr/lib/lpf:lf=/usr/adm/lpd-errs:
```

The *lp* entry specifies the file name to open for output. Here it could be left out since *"/dev/lp"* is the default. The *br* entry sets the baud rate for the tty line and the *fs* entry sets CRMOD, no parity, and XTABS (see *tty(4)*). The *tr* entry indicates that a form-feed should be printed when the queue empties so the paper can be torn off without turning the printer off-line and pressing form feed. The *of* entry specifies the filter program *lpf* should be used for printing the files; more will be said about filters later. The last entry causes errors to be written to the file *"/usr/adm/lpd-errs"* instead of the console. Most errors from *lpd* are logged using *syslogd(8)* and will not be logged in the specified file. The filters should use *syslogd* to report errors; only those that write to standard error output will end up with errors in the *lf* file. (Occasionally errors sent to standard error output have not appeared in the log file; the use of *syslogd* is highly recommended.)



#### 4.1.2. Remote printers

Printers that reside on remote hosts should have an empty `lp` entry. For example, the following `printcap` entry would send output to the printer named “lp” on the machine “ucbvax”.

```
lp|default line printer:\
:lp=:rm=ucbvax:rp=lp:sd=/usr/spool/vaxlpd:
```

The `rm` entry is the name of the remote machine to connect to; this name must be a known host name for a machine on the network. The `rp` capability indicates the name of the printer on the remote machine is “lp”; here it could be left out since this is the default value. The `sd` entry specifies “/usr/spool/vaxlpd” as the spooling directory instead of the default value of “/usr/spool/lpd”.

#### 4.2. Output filters

Filters are used to handle device dependencies and to do accounting functions. The output filtering of `of` is used when accounting is not being done or when all text data must be passed through a filter. It is not intended to do accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owners’ login name, identifying the beginning and ending of jobs, etc. The other filters (if specified) are started for each file printed and do accounting if there is an `af` entry. If entries for both of and other filters are specified, the output filter is used only to print the banner page; it is then stopped to allow other filters access to the printer. An example of a printer that requires output filters is the Benson-Varian.

```
va|varian|Benson-Varian:\
:lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
:tf=/usr/lib/rvcac:mx#2000:pl#58:px=2112:py=1700:tr=\f:
```

The `tf` entry specifies “/usr/lib/rvcac” as the filter to be used in printing *troff*(1) output. This filter is needed to set the device into print mode for text, and plot mode for printing *troff* files and raster images (see *va*(4V)). Note that the page length is set to 58 lines by the `pl` entry for 8.5” by 11” fan-fold paper. To enable accounting, the `varian` entry would be augmented with an `af` filter as shown below.

```
va|varian|Benson-Varian:\
:lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
:if=/usr/lib/vpf:tf=/usr/lib/rvcac:af=/usr/adm/vaacct:\
:mx#2000:pl#58:px=2112:py=1700:tr=\f:
```

#### 4.3. Access Control

Local access to printer queues is controlled with the `rg` `printcap` entry.

```
:rg=lprgroup:
```

Users must be in the group *lprgroup* to submit jobs to the specified printer. The default is to allow all users access. Note that once the files are in the local queue, they can be printed locally or forwarded to another host depending on the configuration.

Remote access is controlled by listing the hosts in either the file */etc/hosts.equiv* or */etc/hosts.lpd*, one host per line. Note that *rsh*(1) and *rlogin*(1) use */etc/hosts.equiv* to determine which hosts are equivalent for allowing logins without passwords. The file */etc/hosts.lpd* is only used to control which hosts have line printer access. Remote access can be further restricted to only allow remote users with accounts on the local host to print jobs by using the `rs` `printcap` entry.

```
:rs:
```



## 5. Output filter specifications

The filters supplied with 4.3BSD handle printing and accounting for most common line printers, the Benson-Varian, the wide (36") and narrow (11") Versatec printer/plotters. For other devices or accounting methods, it may be necessary to create a new filter.

Filters are spawned by *lpd* with their standard input the data to be printed, and standard output the printer. The standard error is attached to the *lf* file for logging errors or *syslogd* may be used for logging errors. A filter must return a 0 exit code if there were no errors, 1 if the job should be reprinted, and 2 if the job should be thrown away. When *lprm* sends a kill signal to the *lpd* process controlling printing, it sends a SIGINT signal to all filters and descendants of filters. This signal can be trapped by filters that need to do cleanup operations such as deleting temporary files.

Arguments passed to a filter depend on its type. The *of* filter is called with the following arguments.

```
filter -wwidth -llength
```

The *width* and *length* values come from the *pw* and *pl* entries in the *printcap* database. The *if* filter is passed the following parameters.

```
filter [-c] -wwidth -llength -lindent -n login -h host accounting_file
```

The *-c* flag is optional, and only supplied when control characters are to be passed uninterpreted to the printer (when using the *-l* option of *lpr* to print the file). The *-w* and *-l* parameters are the same as for the *of* filter. The *-n* and *-h* parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from *printcap*.

All other filters are called with the following arguments:

```
filter -xwidth -ylength -n login -h host accounting_file
```

The *-x* and *-y* options specify the horizontal and vertical page size in pixels (from the *px* and *py* entries in the *printcap* file). The rest of the arguments are the same as for the *if* filter.

## 6. Line printer Administration

The *lpc* program provides local control over line printer activity. The major commands and their intended use will be described. The command format and remaining commands are described in *lpc(8)*.

### abort and start

*Abort* terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by *lpr*). This is normally used to forcibly restart a hung line printer daemon (i.e., *lpq* reports that there is a daemon present but nothing is happening). It does not remove any jobs from the queue (use the *lprm* command instead). *Start* enables printing and requests *lpd* to start printing jobs.

### enable and disable

*Enable* and *disable* allow spooling in the local queue to be turned on/off. This will allow/prevent *lpr* from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the *root* user can still use *lpr* to put jobs in the queue but no one else can. The other main use is to prevent users from putting jobs in the queue when the printer is expected to be unavailable for a long time.

### restart

*Restart* allows ordinary users to restart printer daemons when *lpq* reports that there is no daemon present.

### stop

*Stop* halts a spooling daemon after the current job completes; this also disables printing. This is a clean way to shutdown a printer to do maintenance, etc. Note that users can still enter jobs in a spool queue while a printer is *stopped*.



**topq**

*Topq* places jobs at the top of a printer queue. This can be used to reorder high priority jobs since *lpr* only provides first-come-first-serve ordering of jobs.

**7. Troubleshooting**

There are several messages that may be generated by the the line printer system. This section categorizes the most common and explains the cause for their generation. Where the message implies a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer from the *printcap* database.

**7.1. LPR*****lpr: printer: unknown printer***

The *printer* was not found in the *printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the */etc/printcap* file.

***lpr: printer: jobs queued, but cannot start daemon.***

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket */dev/printer* to be sure it still exists (if it does not exist, there is no *lpd* process running). Usually it is enough to get a super-user to type the following to restart *lpd*.

```
% /usr/lib/lpd
```

You can also check the state of the master printer daemon with the following.

```
% ps l'cat /usr/spool/lpd.lock'
```

Another possibility is that the *lpr* program is not set-user-id to *root*, set-group-id to group *daemon*. This can be checked with

```
% ls -lg /usr/ucb/lpr
```

***lpr: printer: printer queue is disabled***

This means the queue was turned off with

```
% lpc disable printer
```

to prevent *lpr* from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with *lpc*.

**7.2. LPQ****waiting for *printer* to become ready (offline ?)**

The printer device could not be opened by the daemon. This can happen for several reasons, the most common is that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, etc. The actual reason is dependent on the meaning of error codes returned by system device driver. Not all printers supply enough information to distinguish when a printer is off-line or having trouble (e.g. a printer connected through a serial line). Another possible cause of this message is some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with *lpc*.

**printer is ready and printing**

The *lpq* program checks to see if a daemon process exists for *printer* and prints the file *status* located in the spooling directory. If the daemon is hung, a super user can use *lpc* to abort the current daemon and start a new one.

**waiting for host to come up**

This implies there is a daemon trying to connect to the remote machine named *host* to send the files in the local queue. If the remote machine is up, *lpd* on the remote machine is probably dead or hung and should be restarted as mentioned for *lpr*.

**sending to host**

The files should be in the process of being transferred to the remote *host*. If not, the local daemon should be aborted and started with *lpc*.

**Warning: printer is down**

The printer has been marked as being unavailable with *lpc*.

**Warning: no daemon present**

The *lpd* process overseeing the spooling queue, as specified in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer and the *syslogd* logs should be checked for a diagnostic from the deceased process. To restart an *lpd*, use

% *lpc* restart *printer*

**no space on remote; waiting for queue to drain**

This implies that there is insufficient disk space on the remote. If the file is large enough, there will never be enough space on the remote (even after the queue on the remote is empty). The solution here is to move the spooling queue or make more free space on the remote.

**7.3. LPRM****lprm: printer: cannot restart printer daemon**

This case is the same as when *lpr* prints that the daemon cannot be started.

**7.4. LPD**

The *lpd* program can log many different messages using *syslogd*(8). Most of these messages are about files that can not be opened and usually imply that the *printcap* file or the protection modes of the files are incorrect. Files may also be inaccessible if people manually manipulate the line printer system (i.e. they bypass the *lpr* program).

In addition to messages generated by *lpd*, any of the filters that *lpd* spawns may log messages using *syslogd* or to the error log file (the file specified in the *lf* entry in *printcap*).

**7.5. LPC****couldn't start printer**

This case is the same as when *lpr* reports that the daemon cannot be started.



**cannot examine spool directory**

Error messages beginning with "cannot ..." are usually because of incorrect ownership or protection mode of the lock file, spooling directory or the *lpc* program.

# SENDMAIL

## INSTALLATION AND OPERATION GUIDE

Eric Allman  
Britton-Lee, Inc.

Version 5.8

*Sendmail* implements a general purpose internetwork mail routing facility under the UNIX\* operating system. It is not tied to any one transport protocol — its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail - An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

---

\*UNIX is a trademark of Bell Laboratories.

## TABLE OF CONTENTS

1. BASIC INSTALLATION .....	4
1.1. Off-The-Shelf Configurations .....	4
1.2. Installation Using the Makefile .....	5
1.3. Installation by Hand .....	5
1.3.1. lib/libsys.a .....	5
1.3.2. /usr/lib/sendmail .....	5
1.3.3. /usr/lib/sendmail.cf .....	6
1.3.4. /usr/ucb/newaliases .....	6
1.3.5. /usr/spool/mqueue .....	6
1.3.6. /usr/lib/aliases* .....	6
1.3.7. /usr/lib/sendmail.fc .....	6
1.3.8. /etc/rc .....	6
1.3.9. /usr/lib/sendmail.hf .....	7
1.3.10. /usr/lib/sendmail.st .....	7
1.3.11. /usr/ucb/newaliases .....	7
1.3.12. /usr/ucb/mailq .....	7
2. NORMAL OPERATIONS .....	7
2.1. Quick Configuration Startup .....	7
2.2. The System Log .....	8
2.2.1. Format .....	8
2.2.2. Levels .....	8
2.3. The Mail Queue .....	8
2.3.1. Printing the queue .....	8
2.3.2. Format of queue files .....	8
2.3.3. Forcing the queue .....	9
2.4. The Alias Database .....	10
2.4.1. Rebuilding the alias database .....	10
2.4.2. Potential problems .....	11
2.4.3. List owners .....	11
2.5. Per-User Forwarding (.forward Files) .....	11
2.6. Special Header Lines .....	12
2.6.1. Return-Receipt-To: .....	12
2.6.2. Errors-To: .....	12
2.6.3. Apparently-To: .....	12
3. ARGUMENTS .....	12
3.1. Queue Interval .....	12
3.2. Daemon Mode .....	12
3.3. Forcing the Queue .....	12
3.4. Debugging .....	12
3.5. Trying a Different Configuration File .....	13
3.6. Changing the Values of Options .....	13
4. TUNING .....	13
4.1. Timeouts .....	13

4.1.1. Queue interval .....	14
4.1.2. Read timeouts .....	14
4.1.3. Message timeouts .....	14
4.2. Forking During Queue Runs .....	14
4.3. Queue Priorities .....	14
4.4. Load Limiting .....	15
4.5. Delivery Mode .....	15
4.6. Log Level .....	15
4.7. File Modes .....	16
4.7.1. To suid or not to suid? .....	16
4.7.2. Temporary file modes .....	16
4.7.3. Should my alias database be writable? .....	16
5. THE WHOLE SCOOP ON THE CONFIGURATION FILE .....	16
5.1. The Syntax .....	17
5.1.1. R and S — rewriting rules .....	17
5.1.2. D — define macro .....	17
5.1.3. C and F — define classes .....	17
5.1.4. M — define mailer .....	18
5.1.5. H — define header .....	18
5.1.6. O — set option .....	18
5.1.7. T — define trusted users .....	19
5.1.8. P — precedence definitions .....	19
5.2. The Semantics .....	19
5.2.1. Special macros, conditionals .....	19
5.2.2. Special classes .....	21
5.2.3. The left hand side .....	21
5.2.4. The right hand side .....	21
5.2.5. Semantics of rewriting rule sets .....	22
5.2.6. Mailer flags etc. ....	23
5.2.7. The “error” mailer .....	23
5.3. Building a Configuration File From Scratch .....	23
5.3.1. What you are trying to do .....	23
5.3.2. Philosophy .....	24
5.3.2.1. Large site, many hosts — minimum information .....	24
5.3.2.2. Small site — complete information .....	25
5.3.2.3. Single host .....	25
5.3.3. Relevant issues .....	25
5.3.4. How to proceed .....	25
5.3.5. Testing the rewriting rules — the -bt flag .....	26
5.3.6. Building mailer descriptions .....	26
Appendix A. COMMAND LINE FLAGS .....	29
Appendix B. CONFIGURATION OPTIONS .....	30
Appendix C. MAILER FLAGS .....	32
Appendix D. OTHER CONFIGURATION .....	34
Appendix E. SUMMARY OF SUPPORT FILES .....	38



## 1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are acceptable. All pathnames and examples are given from the root of the *sendmail* subtree, normally */usr/src/usr.lib/sendmail* on 4.3BSD.

### 1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectories *cf.named* and *cf.hosttable* of the sendmail directory. The directory *cf.named* contains configuration files that have been tailored for the name server *named*(8). These are the configuration files currently being used at Berkeley. The configuration files in *cf.hosttable* are some typical ones and the old Berkeley versions from before the name server was being used. You should create a symbolic link from *cf* to the directory that you are going to use. For example, to use the name server:

```
ln -s cf.named cf
```

The ones used at Berkeley are in *m4*(1) format; files with names ending “.m4” are *m4* include files, while files with names ending “.mc” are the master files. Files with names ending “.cf” are the *m4* processed versions of the corresponding “.mc” file.

Three off the shelf configurations are supplied to handle the basic cases:

- (1) Arpanet (TCP) sites not running the name server can use *cf.hosttable/arpaproto.cf*. For simple sites, you should be able to use this file without modification. This file is not in *m4* format.
- (2) UUCP sites can use *cf.hosttable/uucpproto.cf*. If your UUCP node name is not the same as your system name (as printed by the *hostname*(1) command) you may have to modify the U macro. This file is not in *m4* format.
- (3) A group of machines at a single site connected by an ethernet with (only) one host connected to the outside world via UUCP is represented by two configuration files: *cf.hosttable/lanroot.mc* should be installed on the host with outside connections and *cf.hosttable/lanleaf.mc* should be installed on all other hosts. These will require slightly more configuration. First, in both files the **D** macro and **D** class must be adjusted to indicate your local domain. For example, if your company is known as “Muse” you will want to change both of those accordingly. (As distributed, they are called XXX.) Second, in *lanleaf.mc* you will have to change the **R** macro to the name of the root host, that is, the host that runs *lanroot.mc*. For example, they might appear as:

```
DDMuse
CDLOCAL Muse
DRErato
```

Internally, the root host will be known as “Erato.Muse” and other hosts will be known as “Thalia.Muse”, “Clio.Muse”, etc.

The file you need should be copied to a file with the same name as your system, e.g.,



```
cp uucppproto.cf ucsfctl.cf
```

This file is now ready for installation as `/usr/lib/sendmail.cf`.

### 1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.3BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should create a symbolic link from *cf* to the directory containing your configuration files. You should also have created your configuration file and left it in the file "*cf/system.cf*" where *system* is the name of your system (i.e., what is returned by *hostname(1)*). If you do not have *hostname* you can use the declaration "*HOST=system*" on the *make(1)* command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

```
make
make install
make installcf
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second and third *make* commands must be executed as the superuser (root).

### 1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

#### 1.3.1. lib/libsys.a

The library in *lib/libsys.a* contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the 4.3BSD directory code and do not have the compatibility routines installed in your system library, you should execute the command:

```
(cd lib; make ndir)
```

This will compile and install the 4.3 compatibility routines in the library. You should then type:

```
(cd lib; make)
```

This will recompile and fill the library.

#### 1.3.2. /usr/lib/sendmail

The binary for *sendmail* is located in */usr/lib*. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:



```
cd src
make clean
make
cp sendmail /usr/lib
chgrp kmem /usr/lib/sendmail
```

### 1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

### 1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

### 1.3.5. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run *setuid*, when *mqueue* should be owned by the sendmail owner and mode 755.

### 1.3.6. /usr/lib/aliases\*

The system aliases are held in three files. The file “/usr/lib/aliases” is the master copy. A sample is given in “lib/aliases” which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm(3)* routines. These are stored in “/usr/lib/aliases.dir” and “/usr/lib/aliases.pag.” These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

### 1.3.7. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

### 1.3.8. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for



connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to “/etc/rc” (or “/etc/rc.local” as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
    (cd /usr/spool/mqueue; rm -f [lnx]**)
    /usr/lib/sendmail -bd -q30m &
    echo -n ' sendmail' >/dev/console
fi
```

The “cd” and “rm” commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: “-bd” causes it to listen on the SMTP port, and “-q30m” causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the -bd flag.

### 1.3.9. /usr/lib/sendmail.hf

This is the help file used by the SMTP **HELP** command. It should be copied from “lib/sendmail.hf”:

```
cp lib/sendmail.hf /usr/lib
```

### 1.3.10. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file “usr/lib/sendmail.st”:

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program “aux/mailstats.”

### 1.3.11. /usr/ucb/newaliases

If *sendmail* is invoked as “newaliases,” it will simulate the -bi flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

### 1.3.12. /usr/ucb/mailq

If *sendmail* is invoked as “mailq,” it will simulate the -bp flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

## 2. NORMAL OPERATIONS

### 2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the -bz flag:

```
/usr/lib/sendmail -bz
```

This creates the file */usr/lib/sendmail.fc* (“frozen configuration”). This file is an image of *sendmail*’s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.



The frozen configuration file will be ignored if a `-C` flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

## 2.2. The System Log

The system log is supported by the *syslogd*(8) program.

### 2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

### 2.2.2. Levels

If you have *syslogd*(8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.6.

## 2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

### 2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the `-bp` flag to sendmail):

```
mailq
```

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

### 2.3.2. Format of queue files

All queue files have the form *xfAA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

- d The data file. The message body (excluding the header) is kept in this file.
- l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous *lf* file can cause a job to apparently disappear (it will not even time out!).
- n This file is created when an *id* is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.
- q The queue control file. This file contains the information necessary to process the job.



- t A temporary file. These are an image of the `qf` file when it is being rebuilt. It should be renamed to a `qf` file very quickly.
- x A transcript file, existing during the life of a session showing everything that happens during that session.

The `qf` file is structured as a series of lines each beginning with a code letter. The lines are as follows:

- D The name of the data file. There may only be one of these lines.
- H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.
- R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.
- S The sender address. There may only be one of these lines.
- E An error address. If any such lines exist, they represent the addresses that should receive error messages.
- T The job creation time. This is used to compute when to time out the job.
- P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority changes as the message sits in the queue. The initial priority depends on the message class and the size of the message.
- M A message. This line is printed by the `mailq` command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to “`mckusick@calder`” and “`wnj`”:

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
          id A13557; 23-Oct-82 15:49:32-PDT (Sat)
HTto: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

### 2.3.3. Forcing the queue

*Sendmail* should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The `-oQ` flag specifies an alternate queue directory and the `-q` flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the `-v` flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

## 2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

```
name: name1, name2, ...
```

Only local names may be aliased; e.g.,

```
eric@mit-xx: eric@berkeley.EDU
```

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign (“#”) are comments.

The second form is processed by the *dbm(3)* library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

### 2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

```
newaliases
```

This is equivalent to giving *sendmail* the `-bi` flag:

```
/usr/lib/sendmail -bi
```

If the “D” option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under

which it will do this are:

- (1) The DBM version of the database is mode 666. -or-
- (2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

#### 2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before *sendmail* will access the database, it checks to insure that this entry exists<sup>1</sup>. *Sendmail* will wait for this entry to appear, at which point it will force a rebuild itself<sup>2</sup>.

#### 2.4.3. List owners

If an error occurs on sending to a certain address, say "x", *sendmail* will look for an alias of the form "owner-x" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintenance of the list itself; in this case the list maintainer would be the owner of the list. For example:

```
unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
               sam@matisse
owner-unix-wizards: eric@ucbarpa
```

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

#### 2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

```
mckusick@ernie
kirk@calder
```

then any mail arriving for "mckusick" will be redirected to the specified accounts.

<sup>1</sup>The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

<sup>2</sup>Note: the "D" option must be specified in the configuration file for this operation to occur. If the "D" option is not specified, a warning message is generated and *sendmail* continues.

## 2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

### 2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete, that is, when successfully delivered to a mailer with the *l* flag (local delivery) set in the mailer descriptor.

### 2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

### 2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

## 3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

### 3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the *-q* flag. If you run in mode *f* or *a* this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in *q* mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

### 3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the *-bd* flag. The *-bd* flag and the *-q* flag may be combined in one call:

```
/usr/lib/sendmail -bd -q30m
```

### 3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the *-q* flag (with no value). It is entertaining to use the *-v* flag (verbose) when this is done to watch what happens:

```
/usr/lib/sendmail -q -v
```

### 3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more



information. The convention is that levels greater than nine are “absurd,” i.e., they print out so much information that you wouldn’t normally want to see them except for debugging that particular piece of code. Debug flags are set using the `-d` option; the syntax is:

```
debug-flag:  -d debug-list
debug-list:  debug-option [ , debug-option ]
debug-option: debug-range [ . debug-level ]
debug-range: integer | integer - integer
debug-level: integer
```

where spaces are for reading ease only. For example,

```
-d12      Set flag 12 to level 1
-d12.3    Set flag 12 to level 3
-d3-17    Set flags 3 through 17 to level 1
-d3-17.4  Set flags 3 through 17 to level 4
```

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

### 3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the `-C` flag; for example,

```
/usr/lib/sendmail -Ctest.cf
```

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the `-C` flag has no value it defaults to *sendmail.cf* in the current directory.

### 3.6. Changing the Values of Options

Options can be overridden using the `-o` flag. For example,

```
/usr/lib/sendmail -oT2m
```

sets the T (timeout) option to two minutes for this run only.

## 4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line “OT3d” sets option “T” to the value “3d” (three days).

Most of these options default appropriately for most sites. However, sites having very high mail loads may find they need to tune them as appropriate for their mail load. In particular, sites experiencing a large number of small messages, many of which are delivered to many recipients, may find that they need to adjust the parameters dealing with queue priorities.

### 4.1. Timeouts

All time intervals are set using a scaled syntax. For example, “10m” represents ten minutes, whereas “2h30m” represents two and a half hours. The full set of scales is:



s seconds  
 m minutes  
 h hours  
 d days  
 w weeks

#### 4.1.1. Queue interval

The argument to the `-q` flag specifies how often a subdaemon will run the queue. This is typically set to between fifteen minutes and one hour.

#### 4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the `r` option in the configuration file.

#### 4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the `T` option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

```
/usr/lib/sendmail -oTld -q
```

will run the queue and flush anything that is one day old.

### 4.2. Forking During Queue Runs

By setting the `Y` option, *sendmail* will fork before each individual message while running the queue. This will prevent *sendmail* from consuming large amounts of memory, so it may be useful in memory-poor environments. However, if the `Y` option is not set, *sendmail* will keep track of hosts that are down during a queue run, which can improve performance dramatically.

### 4.3. Queue Priorities

Every message is assigned a priority when it is first instantiated, consisting of the message size (in bytes) offset by the message class times the “work class factor” and the number of recipients times the “work recipient factor.” The priority plus the creation time of the message (in seconds since January 1, 1970) are used to order the queue. Higher numbers for the priority mean that the message will be processed later when running the queue.

The message size is included so that large messages are penalized relative to small messages. The message class allows users to send “high priority” messages by including a “Precedence:” field in their message; the value of this field is looked up in the `P` lines of the configuration file. Since the number of recipients affects the amount of load a message presents to the system, this is also included into the priority.

The recipient and class factors can be set in the configuration file using the *y* and *z* options respectively. They default to 1000 (for the recipient factor) and 1800 (for the class factor). The initial priority is:

$$pri = size - (class * z) + (nrpt * y)$$

(Remember, higher values for this parameter actually mean that the job will be treated with lower priority.)

The priority of a job can also be adjusted each time it is processed (that is, each time an attempt is made to deliver it) using the “work time factor,” set by the *Z* option. This is added to the priority, so it normally decreases the precedence of the job, on the grounds that jobs that have failed many times will tend to fail again in the future.

#### 4.4. Load Limiting

*Sendmail* can be asked to queue (but not deliver) mail if the system load average gets too high using the *x* option. When the load average exceeds the value of the *x* option, the delivery mode is set to *q* (queue only) if the *Queue Factor* (*q* option) divided by the difference in the current load average and the *x* option plus one exceeds the priority of the message — that is, the message is queued iff:

$$pri > \frac{QF}{LA - x + 1}$$

The *q* option defaults to 10000, so each point of load average is worth 10000 priority points (as described above, that is, bytes + seconds + offsets).

For drastic cases, the *X* option defines a load average at which *sendmail* will refuse to accept network connections. Locally generated mail (including incoming UUCP mail) is still accepted.

#### 4.5. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the “*d*” configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

- i* deliver interactively (synchronously)
- b* deliver in background (asynchronously)
- q* queue only (don’t deliver)

There are tradeoffs. Mode “*i*” passes the maximum amount of information to the sender, but is hardly ever necessary. Mode “*q*” puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode “*b*” is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

#### 4.6. Log Level

The level of logging can be set for *sendmail*. The default using a standard configuration table is level 9. The levels are as follows:

- 0 No logging.
- 1 Major problems only.
- 2 Message collections and failed deliveries.
- 3 Successful deliveries.



- 4 Messages being deferred (due to a host being down, etc.).
- 5 Normal message queueups.
- 6 Unusual but benign incidents, e.g., trying to process a locked queue file.
- 9 Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.
- 12 Several messages that are basically only of interest when debugging.
- 16 Verbose information regarding the queue.

#### 4.7. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

##### 4.7.1. To suid or not to suid?

*Sendmail* can safely be made setuid to root. At the point where it is about to *exec(2)* a mailer, it checks to see if the *userid* is zero; if so, it resets the *userid* and *groupid* to a default (set by the *u* and *g* options). (This can be overridden by setting the *S* flag to the mailer for mailers that are trusted and must be called as root.) However, this will cause mail processing to be accounted (using *sa(8)*) to root rather than to the user sending the mail.

##### 4.7.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the “F” option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

##### 4.7.3. Should my alias database be writable?

At Berkeley we have the alias database (*/usr/lib/aliases\**) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can “steal” any other user’s mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in */usr/lib*). The mode on these files should match the mode on */usr/lib/aliases*. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the “D” option), then you must be careful to reconstruct the alias database each time you change the text version:

```
newaliases
```

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

## 5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the “future project” list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

### 5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol (*#*) are comments.

#### 5.1.1. R and S — rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by other rewriting sets.

The syntax of these two commands are:

**S*n***

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

**R*lhs rhs comments***

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

#### 5.1.2. D — define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

**D*x val***

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence *\$x*.

#### 5.1.3. C and F — define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

```
Cc word1 word2...
Fc file
```

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

```
CHmonet ucbmonet
```

and

```
CHmonet
CHucbmonet
```

are equivalent. The second form reads the elements of the class *c* from the named *file*.

#### 5.1.4. M — define mailer

Programs and interfaces to mailers are defined in this line. The format is:

```
Mname, (field=value)*
```

where *name* is the name of the mailer (used internally only) and the “field=name” pairs define attributes of the mailer. Fields are:

Path	The pathname of the mailer
Flags	Special flags for this mailer
Sender	A rewriting set for sender addresses
Recipient	A rewriting set for recipient addresses
Argv	An argument vector to pass to this mailer
Eol	The end-of-line string for this mailer
Maxsize	The maximum message length to this mailer

Only the first character of the field name is checked.

#### 5.1.5. H — define header

The format of the header lines that sendmail inserts into the message are defined by the H line. The syntax of this line is:

```
H[?mflags?]{hname: htemplate
```

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

#### 5.1.6. O — set option

There are a number of “random” options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

```
Oo value
```

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values “t”, “T”, “f”, or “F”; the default is TRUE), or a time interval.

### 5.1.7. T — define trusted users

Trusted users are those users who are permitted to override the sender address using the `-f` flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

```
Tuser1 user2...
```

There may be more than one of these lines.

### 5.1.8. P — precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

```
Pname=num
```

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

```
Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100
```

## 5.2. The Semantics

This section describes the semantics of the configuration file.

### 5.2.1. Special-macros, conditionals

Macros are interpolated using the construct `$x`, where *x* is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

Conditionals can be specified using the syntax:

```
$?x text1 $| text2 $.
```

This interpolates *text1* if the macro `$x` is set, and *text2* otherwise. The "else" (`$|`) clause may be omitted.

The following macros *must* be defined to transmit information into *sendmail*:

- e The SMTP entry message
- j The "official" domain name for this site
- l The format of the UNIX from line
- n The name of the daemon (for error messages)
- o The set of "operators" in addresses
- q default format of sender address

The `$e` macro is printed out when SMTP starts up. The first word must be the `$j` macro. The `$j` macro should be in RFC821 format. The `$l` and `$n` macros can be considered constants except under terribly unusual circumstances. The `$o` macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "@" were in the `$o` macro, then the input "a@b" would be scanned as three tokens: "a," "@," and "b." Finally, the `$q` macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:



```

De$J Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g $d
Do.:%@!"/
Dq$g$?x ($x)$
Dj$H.$D

```

An acceptable alternative for the \$q macro is "\$?x\$x \$.<\$g>". These correspond to the following two formats:

```

eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>

```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

- a The origination date in Arpanet format
- b The current date in Arpanet format
- c The hop count
- d The date in UNIX (ctime) format
- f The sender (from) address
- g The sender address relative to the recipient
- h The recipient host
- i The queue id
- p Sendmail's pid
- r Protocol used
- s Sender's host name
- t A numeric representation of the current time
- u The recipient user
- v The version number of sendmail
- w The hostname of this site
- x The full name of the sender
- z The home directory of the recipient

There are three types of dates that can be used. The \$a and \$b macros are in Arpanet format; \$a is the time as extracted from the "Date:" line of the message (if there was one), and \$b is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, \$a is set to the current time also. The \$d macro is equivalent to the \$a macro in UNIX (ctime) format.

The \$f macro is the id of the sender as originally determined; when mailing to a specific host the \$g macro is set to the address of the sender *relative to the recipient*. For example, if I send to "bollard@matisse" from the machine "ucbarpa" the \$f macro will be "eric" and the \$g macro will be "eric@ucbarpa."

The \$x macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail*. The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the \$h, \$u, and \$z macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the \$@ and \$: part of the rewriting rules, respectively.

The \$p and \$t macros are used to create unique strings (e.g., for the "Message-Id:" field). The \$i macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The \$v macro



is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The `$w` macro is set to the name of this host if it can be determined. The `$c` field is set to the “hop count,” i.e., the number of times this message has been processed. This can be determined by the `-h` flag on the command line or by counting the timestamps in the message.

The `$r` and `$s` fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

### 5.2.2. Special classes

The class `$=w` is set to be the set of all names this host is known by. This can be used to delete local hostnames.

### 5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasympols are:

```
$*   Match zero or more tokens
$+   Match one or more tokens
$-   Match exactly one token
$=x  Match any token in class x
$~x  Match any token not in class x
```

If any of these match, they are assigned to the symbol `$n` for replacement on the right hand side, where `n` is the index in the LHS. For example, if the LHS:

```
$-:$+
```

is applied to the input:

```
UCBARPA:eric
```

the rule will match, and the values passed to the RHS will be:

```
$1 UCBARPA
$2 eric
```

### 5.2.4. The right hand side

When the left hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they begin with a dollar sign. Metasympols are:

```
$n      Substitute indefinite token n from LHS
${name$} Canonicalize name
$>n     “Call” ruleset n
$#mailer Resolve to mailer
$@host  Specify host
$:user  Specify user
```

The `$n` syntax substitutes the corresponding value from a `$+`, `$-`, `$*`, `$=`, or `$~` match on the LHS. It may be used anywhere.

A host name enclosed between `$(` and `$)` is looked up using the *gethostent*(3) routines and replaced by the canonical name. For example, `“${csam$}”` would become `“lbl-csam.arpa”` and `“${[128.32.130.2]$}”` would become `“vangogh.berkeley.edu.”`



The  $\$>n$  syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset  $n$ . The final value of ruleset  $n$  then becomes the substitution for this rule.

The  $\#\#$  syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

$\#\#\text{mailer}\$@\text{host}\$:user$

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceded by a  $\$@$  or a  $\$:$  to control evaluation. A  $\$@$  prefix causes the ruleset to return with the remainder of the RHS as the value. A  $\$:$  prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The  $\$@$  and  $\$:$  prefixes may precede a  $\$>$  spec; for example:

$R\$+ \quad \$:\$>7\$1$

matches anything, passes that to ruleset seven, and continues; the  $\$:$  is necessary to avoid an infinite loop.

Substitution occurs in the order described, that is, parameters from the LHS are substituted, hostnames are canonicalized, "subroutines" are called, and finally  $\#\#$ ,  $\$@$ , and  $\$:$  are processed.

#### 5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

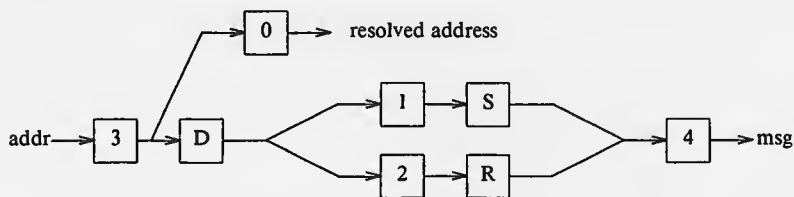


Figure 2 — Rewriting set semantics  
 D — sender domain addition  
 S — mailer-specific sender rewriting  
 R — mailer-specific recipient rewriting

local-part@host-domain-spec

If no “@” sign is specified, then the host-domain-spec *may* be appended from the sender address (if the C flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the \$h macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

#### 5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

#### 5.2.7. The “error” mailer

The mailer with the special name “error” can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

```
$#error$:Host unknown in this domain
```

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

### 5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

#### 5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally,



ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

### 5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

#### 5.3.2.1. Large site, many hosts — minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts and multiple mail connections. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." When monet receives mail for delivery, it checks whether it knows that the destination host is directly reachable; if so, mail is sent to that host. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new mail connection is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated if convenient, but this is not critical.

This picture is slightly muddled due to network connections that are not actually located on ucbvax. For example, some UUCP connections are currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a UUCP host is transferred via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to UUCP. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send appropriate UUCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as connected to ucbarpa it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, for example, if ucbarpa thought that ucbvax had the UUCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4(1)* or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

### 5.3.2.2. Small site — complete information

A small site (two or three hosts and few external connections) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the configuration remains relatively static, the update problem will probably not be too great.

### 5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

### 5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

`< > ( ) " \`

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host might be "a.CC.Berkeley.EDU"; reading from right to left, "EDU" is a top level domain comprising educational institutions, "Berkeley" is a logical domain name, "CC" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center.

Beware when reading RFC819 that there are a number of errors in it.

### 5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define



ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

### 5.3.5. Testing the rewriting rules — the `-bt` flag

When you build a configuration table, you can do a certain amount of testing using the “test mode” of *sendmail*. For example, you could invoke *sendmail* as:

```
sendmail -bt -Ctest.cf
```

which would read the configuration file “test.cf” and enter test mode. In this mode, you enter lines of the form:

```
rwset address
```

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of *rwsets* for sequential application of rules to an input; ruleset three is always applied first. For example:

```
1,21,4 monet:bollard
```

first applies ruleset three to the input “monet:bollard.” Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the “-d21” flag to turn on more debugging. For example,

```
sendmail -bt -d21.99
```

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

### 5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names “local” and “prog” must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string “[IPC]” instead.

The F field defines the mailer flags. You should specify an “f” or “r” flag to pass the name of the sender as a `-f` or `-r` flag respectively. These flags are only passed if they were passed to *sendmail*, so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify “-f \$g” in the argv template. If the mailer must be called as root the “S” flag should be given; this will not reset the userid before calling the mailer<sup>3</sup>. If this mailer is local (i.e., will perform final delivery rather than another network hop) the “l” flag should be given. Quote characters (backslashes and “ marks) can be stripped from addresses if the “s” flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the “m” flag should be stated. If this flag is on, then the argv template

<sup>3</sup>*Sendmail* must be running setuid to root for this to work.

containing \$u will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run<sup>4</sup>.

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa  
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa  
To: wnj@monet, mckusick@ucbarpa

if and only if the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbox!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a \$u macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC \$h [ port ]

where *port* is the optional port number to connect to.

For example, the specifications:

Mlocal, P=/bin/mail, F=rlsm S=10, R=20, A=mail -d \$u  
Mether, P=[IPC], F=mcC, S=11, R=21, A=IPC \$h, M=100000

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky -r flag, does local delivery, quotes should be stripped from addresses, and multiple users can be

<sup>4</sup>The "c" configuration option must be given for this to be effective.



delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a -r flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.



## APPENDIX A

### COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

- f *addr*** The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).
  - r *addr*** An obsolete form of -f.
  - h *cnt*** Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.
  - F*name*** Sets the full name of this user to *name*.
  - n** Don't do aliasing or forwarding.
  - t** Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.
  - bx** Set operation mode to *x*. Operation modes are:
    - m** Deliver mail (default)
    - a** Run in arpanet mode (see below)
    - s** Speak SMTP on input side
    - d** Run as a daemon
    - t** Run in test mode
    - v** Just verify addresses, don't collect or deliver
    - i** Initialize the alias database
    - p** Print the mail queue
    - z** Freeze the configuration file
- The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.
- q*time*** Try to process the queued up mail. If the time is given, a *sendmail* will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.
  - C*file*** Use a different configuration file. *Sendmail* runs as the invoking user (rather than root) when this flag is specified.
  - d*level*** Set debugging level.
  - o*x value*** Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the -s flag.



## APPENDIX B

### CONFIGURATION OPTIONS

The following options may be set using the `-o` flag on the command line or the `O` line in the configuration file. Many of them cannot be specified unless the invoking user is trusted.

<i>Afile</i>	Use the named <i>file</i> as the alias file. If no file is specified, use <i>aliases</i> in the current directory.
<i>aN</i>	If set, wait up to <i>N</i> minutes for an "@:@" entry to exist in the alias database before starting up. If it does not appear in <i>N</i> minutes, rebuild the database (if the <code>D</code> option is also set) or issue a warning.
<i>Bc</i>	Set the blank substitution character to <i>c</i> . Unquoted spaces in addresses are replaced by this character.
<i>c</i>	If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.
<i>dx</i>	Deliver in mode <i>x</i> . Legal modes are: <ul style="list-style-type: none"><li><i>i</i> Deliver interactively (synchronously)</li><li><i>b</i> Deliver in background (asynchronously)</li><li><i>q</i> Just queue the message (deliver during queue run)</li></ul>
<i>D</i>	If set, rebuild the alias database if necessary and possible. If this option is not set, <i>sendmail</i> will never rebuild the alias database unless explicitly requested using <code>-bi</code> .
<i>ex</i>	Dispose of errors using mode <i>x</i> . The values for <i>x</i> are: <ul style="list-style-type: none"><li><i>p</i> Print error messages (default)</li><li><i>q</i> No messages, just give exit status</li><li><i>m</i> Mail back errors</li><li><i>w</i> Write back errors (mail if user not logged in)</li><li><i>e</i> Mail back errors and give zero exit stat always</li></ul>
<i>Fn</i>	The temporary file mode, in octal. 644 and 600 are good choices.
<i>f</i>	Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.
<i>gn</i>	Set the default group id for mailers to run in to <i>n</i> .
<i>Hfile</i>	Specify the help file for SMTP.
<i>i</i>	Ignore dots in incoming messages.
<i>Ln</i>	Set the default log level to <i>n</i> .
<i>Mxvalue</i>	Set the macro <i>x</i> to <i>value</i> . This is intended only for use from the command line.
<i>m</i>	Send to me too, even if I am in an alias expansion.
<i>Nnetname</i>	The name of the home network; "ARPA" by default. The the argument of an SMTP "HELO" command is checked against "hostname.netname" where

	<i>hostname</i> is requested from the kernel for the current connection. If they do not match, "Received:" lines are augmented by the name that is determined in this manner so that messages can be traced accurately.
<b>o</b>	Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.
<b>Qdir</b>	Use the named <i>dir</i> as the queue directory.
<b>qfactor</b>	Use <i>factor</i> as the multiplier in the map function to decide when to just queue up jobs rather than run them. This value is divided by the difference between the current load average and the load average limit ( <i>x</i> flag) to determine the maximum message priority that will be sent. Defaults to 10000.
<b>rtime</b>	Timeout reads after <i>time</i> interval.
<b>Sfile</b>	Log statistics in the named <i>file</i> .
<b>s</b>	Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. <i>Sendmail</i> always instantiates the queue file before returning control the the client under any circumstances.
<b>Ttime</b>	Set the queue timeout to <i>time</i> . After this interval, messages that have not been successfully sent will be returned to the sender.
<b>tS,D</b>	Set the local time zone name to <i>S</i> for standard time and <i>D</i> for daylight time; this is only used under version six.
<b>un</b>	Set the default userid for mailers to <i>n</i> . Mailers without the <i>S</i> flag in the mailer definition will run as this user.
<b>v</b>	Run in verbose mode.
<b>xLA</b>	When the system load average exceeds <i>LA</i> , just queue messages (i.e., don't try to send them).
<b>XLA</b>	When the system load average exceeds <i>LA</i> , refuse incoming SMTP connections.
<b>yfact</b>	The indicated <i>factor</i> is added to the priority (thus <i>lowering</i> the priority of the job) for each recipient, i.e., this value penalizes jobs with large numbers of recipients.
<b>Y</b>	If set, deliver each job that is run from the queue in a separate process. Use this option if you are short of memory, since the default tends to consume considerable amounts of memory while the queue is being processed.
<b>zfact</b>	The indicated <i>factor</i> is multiplied by the message class (determined by the Precedence: field in the user header and the <i>P</i> lines in the configuration file) and subtracted from the priority. Thus, messages with a higher Priority: will be favored.
<b>Zfact</b>	The <i>factor</i> is added to the priority every time a job is processed. Thus, each time a job is processed, its priority will be decreased by the indicated value. In most environments this should be positive, since hosts that are down are all too often down for a long time.



## APPENDIX C

### MAILER FLAGS

The following flags may be set in the mailer description.

- f The mailer wants a `-f from` flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).
- r Same as f, but sends a `-r` flag.
- S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).
- n Do not insert a UNIX-style "From" line on the front of the message.
- l This mailer is local (i.e., final delivery will be performed).
- s Strip quote characters off of the address before calling the mailer.
- m This mailer can send to multiple users on the same host in one transaction. When a `$u` macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.
- F This mailer wants a "From:" header line.
- D This mailer wants a "Date:" header line.
- M This mailer wants a "Message-Id:" header line.
- x This mailer wants a "Full-Name:" header line.
- P This mailer wants a "Return-Path:" line.
- u Upper case should be preserved in user names for this mailer.
- h Upper case should be preserved in host names for this mailer.
- A This is an Arpanet-compatible mailer, and all appropriate modes should be set.
- U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.
- e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.
- X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.
- L Limit the line lengths as specified in RFC821.
- P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.
- l This mailer will be speaking SMTP to another *sendmail* — as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

- C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign (“@”) after being rewritten by ruleset three will have the “@domain” clause from the sender tacked on. This allows mail with headers of the form:

From: usera@hosta  
To: userb@hostb, userc

to be rewritten as:

From: usera@hosta  
To: userb@hostb, userc@hosta

automatically.

- E Escape lines beginning with “From” in the message with a ‘>’ sign.



## APPENDIX D

### OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

- md/config.m4** These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.
- src/conf.h** Configuration parameters that may be tweaked by the installer are included in *conf.h*.
- src/conf.c** Some special routines and a few variables may be defined in *conf.c*. For the most part these are selected from the settings in *conf.h*.

#### Parameters in md/config.m4

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

- V6** If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.
- VMUNIX** If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork(2)* system call, special types defined in *<sys/types.h>* (e.g, *u\_char*), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

#### Parameters in src/conf.h

Parameters and compilation options are defined in *conf.h*. Most of these need not normally be tweaked; common parameters are all in *sendmail.cf*. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

- MAXLINE [1024]** The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.
- MAXNAME [256]** The maximum length of any name, such as a host or a user name.
- MAXFIELD [2500]** The maximum total length of any header field, including continuation lines.
- MAXPV [40]** The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.
- MAXHOP [17]** When a message has been processed more than this number of times, *sendmail* rejects the message on the assumption that there has been an aliasing loop. This can be determined from the *-h* flag or by counting the number

of trace fields (i.e., "Received:" lines) in the message header.

**MAXATOM [100]** The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

**MAXMAILERS [25]**  
The maximum number of mailers that may be defined in the configuration file.

**MAXRWSETS [30]**  
The maximum number of rewriting sets that may be defined.

**MAXPRIORITIES [25]**  
The maximum number of values for the "Precedence:" field that may be defined (using the P line in *sendmail.cf*).

**MAXTRUST [30]** The maximum number of trusted users that may be defined (using the T line in *sendmail.cf*).

**MAXUSERENVIRON [40]**  
The maximum number of items in the user environment that will be passed to subordinate mailers.

**QUEUESIZE [600]**  
The maximum number of entries that will be processed in a single queue run.

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

**DBM** If set, the "DBM" package in UNIX is used (see *dbm(3X)* in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

**NDBM** If set, the new version of the DBM library that allows multiple databases will be used. "DBM" must also be set.

**DEBUG** If set, debugging information is compiled in. To actually get the debugging output, the *-d* flag must be used.

**LOG** If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

**QUEUE** This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

**SMTP** If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

**DAEMON** If set, code to run a daemon is compiled in. This code is for 4.2 or 4.3BSD.

**UGLYUUCP** If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

**NOTUNIX** If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

#### Configuration in *src/conf.c*

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below). The flags are:



H_ACHECK	Normally when the check is made to see if a header line is compatible with a mailer, <i>sendmail</i> will not delete an existing line. If this flag is set, <i>sendmail</i> will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in <i>sendmail.cf</i> , the header line is <i>always</i> deleted.
H_EOH	If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.
H_FORCE	Add this header entry even if one existed in the message before. If a header entry does not have this bit set, <i>sendmail</i> will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.
H_TRACE	If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.
H_RCPT	If set, this field contains recipient addresses. This is used by the <i>-t</i> flag to determine who to send to when it is collecting recipients from the message.
H_FROM	This flag indicates that this field specifies a sender. The order of these fields in the <i>HdrInfo</i> table specifies <i>sendmail's</i> preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```

struct hdrinfo      HdrInfo[] =
(
    /* originator fields, most to least significant */
    "resent-sender",  H_FROM,
    "resent-from",    H_FROM,
    "sender",         H_FROM,
    "from",           H_FROM,
    "full-name",      H_ACHECK,
    /* destination fields */
    "to",             H_RCPT,
    "resent-to",      H_RCPT,
    "cc",             H_RCPT,
    /* message identification and control */
    "message",        H_EOH,
    "text",           H_EOH,
    /* trace fields */
    "received",       H_TRACE|H_FORCE,

    NULL,             0,
);

```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing



performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro \$x and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char Arpa_Info[] = "050"; /* arbitrary info */
char Arpa_TSyserr[] = "455"; /* some (transient) system error */
char Arpa_PSyserr[] = "554"; /* some (permanent) system error */
char Arpa_Usrerr[] = "554"; /* some (fatal) user error */
```

The class *Arpa\_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa\_TSyserr* and *Arpa\_PSyserr* is printed by the *syserr* routine. *TSyserr* is printed out for transient errors, that is, errors that are likely to go away without explicit action on the part of a systems administrator. *PSyserr* is printed for permanent errors. The distinction is made based on the value of *errno*. Finally, *Arpa\_Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q_mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to suppress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

#### Configuration in *src/daemon.c*

The file *src/daemon.c* contains a number of routines that are dependent on the local networking environment. The version supplied is specific to 4.3 BSD.

The routine *maphostname* is called to convert strings within \$[ ... \$] symbols. It can be modified if you wish to provide a more sophisticated service, e.g., mapping UUCP host names to full paths.



## APPENDIX E

### SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

- /usr/lib/sendmail**  
The binary of *sendmail*.
- /usr/bin/newaliases**  
A link to */usr/lib/sendmail*; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **-bi** flag.
- /usr/bin/mailq** Prints a listing of the mail queue. This program is equivalent to using the **-bp** flag to *sendmail*.
- /usr/lib/sendmail.cf**  
The configuration file, in textual form.
- /usr/lib/sendmail.fc**  
The configuration file represented as a memory image.
- /usr/lib/sendmail.hf**  
The SMTP help file.
- /usr/lib/sendmail.st**  
A statistics file; need not be present.
- /usr/lib/aliases** The textual version of the alias file.
- /usr/lib/aliases.{pag,dir}**  
The alias file in *dbm*(3) format.
- /usr/spool/mqueue**  
The directory in which the mail queue and temporary files reside.
- /usr/spool/mqueue/qf\***  
Control (queue) files for messages.
- /usr/spool/mqueue/df\***  
Data files.
- /usr/spool/mqueue/lf\***  
Lock files
- /usr/spool/mqueue/tf\***  
Temporary versions of the *qf* files, used during queue file rebuild.
- /usr/spool/mqueue/nf\***  
A file used when creating a unique id.
- /usr/spool/mqueue/xf\***  
A transcript of the current session.

## Timed Installation and Operation Guide

*Riccardo Gusella, Stefano Zatti, James M. Bloom*

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

*Kirk Smith*

Engineering Computer Network  
Department of Electrical Engineering  
Purdue University  
West Lafayette, IN 47906

### Introduction

The clock synchronization service for the UNIX 4.3BSD operating system is composed of a collection of time daemons (*timed*) running on the machines in a local area network. The algorithms implemented by the service is based on a master-slave scheme. The time daemons communicate with each other using the *Time Synchronization Protocol* (TSP) which is built on the DARPA UDP protocol and described in detail in [4].

A time daemon has a twofold function. First, it supports the synchronization of the clocks of the various hosts in a local area network. Second, it starts (or takes part in) the election that occurs among slave time daemons when, for any reason, the master disappears. The synchronization mechanism and the election procedure employed by the program *timed* are described in other documents [1,2,3]. The next paragraphs are a brief overview of how the time daemon works. This document is mainly concerned with the administrative and technical issues of running *timed* at a particular site.

A *master time daemon* measures the time differences between the clock of the machine on which it is running and those of all other machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks.<sup>1</sup> It then sends to each *slave time daemon* the correction that should be performed on the clock of its machine. This process is repeated periodically. Since the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with the accuracy of the synchronization. When a machine comes up and joins the network, it starts a slave time daemon which will ask the master for the correct time and will reset the machine's clock before any user activity can begin. The time daemons are able to maintain a single network time in spite of the drift of clocks away from each other. The present implementation keeps processor clocks synchronized within 20 milliseconds.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, and by the CSELT Corporation of Italy. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of CSELT.

<sup>1</sup> A clock is considered to be faulty when its value is more than a small specified interval apart from the majority of the clocks of the other machines [1,2].

To ensure that the service provided is continuous and reliable, it is necessary to implement an election algorithm to elect a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned. Under our algorithm, slaves are able to realize when the master has stopped functioning and to elect a new master from among themselves. It is important to note that, since the failure of the master results only in a gradual divergence of clock values, the election need not occur immediately.

The machines that are gateways between distinct local area networks require particular care. A time daemon on such machines may act as a *submaster*. This artifact depends on the current inability of transmission protocols to broadcast a message on a network other than the one to which the broadcasting machine is connected. The submaster appears as a slave on one network, and as a master on one or more of the other networks to which it is connected.

A submaster classifies each network as one of three types. A *slave network* is a network on which the submaster acts as a slave. There can only be one slave network. A *master network* is a network on which the submaster acts as a master. An *ignored network* is any other network which already has a valid master. The submaster tries periodically to become master on an ignored network, but gives up immediately if a master already exists.

### Guidelines

While the synchronization algorithm is quite general, the election one, requiring a broadcast mechanism, puts constraints on the kind of network on which time daemons can run. The time daemon will only work on networks with broadcast capability augmented with point-to-point links. Machines that are only connected to point-to-point, non-broadcast networks may not use the time daemon.

If we exclude submasters, there will normally be, at most, one master time daemon in a local area internetwork. During an election, only one of the slave time daemons will become the new master. However, because of the characteristics of its machine, a slave can be prevented from becoming the master. Therefore, a subset of machines must be designated as potential master time daemons. A master time daemon will require CPU resources proportional to the number of slaves, in general, more than a slave time daemon, so it may be advisable to limit master time daemons to machines with more powerful processors or lighter loads. Also, machines with inaccurate clocks should not be used as masters. This is a purely administrative decision: an organization may well allow all of its machines to run master time daemons.

At the administrative level, a time daemon on a machine with multiple network interfaces, may be told to ignore all but one network or to ignore one network. This is done with the *-n network* and *-i network* options respectively at start-up time. Typically, the time daemon would be instructed to ignore all but the networks belonging to the local administrative control.

There are some limitations to the current implementation of the time daemon. It is expected that these limitations will be removed in future releases. The constant `NHOSTS` in `/usr/src/etc/timed/globals.h` limits the maximum number of machines that may be directly controlled by one master time daemon. The current maximum is 29 (`NHOSTS - 1`). The constant must be changed and the program recompiled if a site wishes to run *timed* on a larger (inter)network.

In addition, there is a *pathological situation* to be avoided at all costs, that might occur when time daemons run on multiply-connected local area networks. In this case, as we have seen, time daemons running on gateway machines will be submasters and they will act on some of those networks as master time daemons. Consider machines A and B that are both gateways between networks X and Y. If time daemons were started on both A and B without constraints, it would be possible for submaster time daemon A to be a slave on network X and the master on network Y, while submaster time daemon B is a slave on network Y and the master on network X. This *loop* of master time daemons will not function properly or guarantee a unique time on both networks, and will cause the submasters to use large amounts of system resources in the form of network bandwidth and CPU time. In fact, this kind of *loop* can also be generated with more than two master time daemons, when several local area networks are interconnected.

### Installation

In order to start the time daemon on a given machine, the following lines should be added to the *local daemons* section in the file */etc/rc.local*:

```
if [ -f /etc/timed ]; then
    /etc/timed flags & echo -n ' timed' >/dev/console
fi
```

In any case, they must appear after the network is configured via *ifconfig*(8).

Also, the file */etc/services* should contain the following line:

```
timed          525/udp          timeserver
```

The *flags* are:

- n network    to consider the named network.
- i network    to ignore the named network.
- t            to place tracing information in */usr/adm/timed.log*.
- M            to allow this time daemon to become a master. A time daemon run without this option will be forced in the state of slave during an election.

### Daily Operation

*Timedc*(8) is used to control the operation of the time daemon. It may be used to:

- measure the differences between machines' clocks,
- find the location where the master *timed* is running,
- cause election timers on several machines to expire at the same time,
- enable or disable tracing of messages received by *timed*.

See the manual page on *timed*(8) and *timedc*(8) for more detailed information.

The *date*(1) command can be used to set the network date. In order to set the time on a single machine, the *-n* flag can be given to *date*(1).

**References**

1. R. Gusella and S. Zatti, *TEMPO: A Network Time Controller for Distributed Berkeley UNIX System*, USENIX Summer Conference Proceedings, Salt Lake City, June 1984.
2. R. Gusella and S. Zatti, *Clock Synchronization in a Local Area Network*, University of California, Berkeley, Technical Report, *to appear*.
3. R. Gusella and S. Zatti, *An Election Algorithm for a Distributed Clock Synchronization Program*, University of California, Berkeley, CS Technical Report #275, Dec. 1985.
4. R. Gusella and S. Zatti, *The Berkeley UNIX 4.3BSD Time Synchronization Protocol*, UNIX Programmer's Manual, 4.3 Berkeley Software Distribution, Volume 2c.

## Installation and Operation of UUCP 4.3BSD Edition

*D. A. Nowitz*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

*Carl S. Gutekunst*

Communications Software Research and Development  
Pyramid Technology Corporation  
Mountain View, California 94039

### ABSTRACT

UUCP is a series of programs designed to permit communication between UNIX<sup>†</sup> systems using a variety of communications links. UUCP provides batched, error free file transfers and remote command execution. It is well suited for tasks such as electronic mail, public news networks, and software distribution, particularly when only slow, low-cost communication links are available (e.g., 1200 baud dial-up).

This document describes the 4.3BSD version of UUCP. This is a distant but direct descendent of the "second implementation" of UUCP developed by D. A. Nowitz at AT&T Bell Laboratories. A number of other UUCP versions are in common usage; these are discussed only to the extent that they affect administration of 4.3BSD systems.

Revised August 24, 1986

### 1. UUCP OVERVIEW

UUCP is a batch-type operation. Users issue commands that are queued in a spool directory for processing by background daemons.

*Uucp* (UNIX-to-UNIX Copy) and *uux* (UNIX-to-UNIX Execution) provide the user interface to UUCP. *Uucp* has syntax and semantics similar to the standard UNIX utility *cp*(1), with the added ability to prefix filenames with system names. Similarly, *uux* mimics the conventions of *sh*(1), and allows commands to be prefixed with system names.

*Uucico* (Copy-In, Copy-Out) is the primary UUCP daemon. It processes the requests queued by *uucp* and *uux*, initiates calls to remote systems, transfers files, and forks other daemons to execute *uux*-requested commands. *Uucico* also acts as the UUCP "shell" when remote systems call in to requests transfers.

Three types of files are used in UUCP operation. *Control files* describe the UUCP environment, including known remote hosts, available devices, and remote file access permissions. Control files are relatively static; they are generally changed only by the system administrator. *Spool files* (also called *Queue files*) contain transfer requests and data; they are created and deleted as necessary by *uucp*, *uux*, and *uucico*. *Log files* accumulate a history of UUCP activity; these tend to grow forever if not

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories.

periodically cleaned up.

Spool files are further divided into three types: *Work files* contain directions for file transfers between systems. Every invocation of *uucp* or *uux* creates one or more work files. *Data files* contain data for transfer to or from remote systems. *Execution files* contain directions for UNIX command executions which involve the resources of one or more systems. Execution files are created only by *uux*.

## 2. USER UTILITIES

UUCP includes a total of ten "primary" utilities, that is, ten utilities for general users. All reside in the */usr/bin* directory, where they are easily accessible. This section provides detailed implementation descriptions for the more important commands; see the corresponding *man* pages for additional information.

The following two commands queue transfer requests:

- uucp(1C)** UNIX-to-UNIX File Copy. One or more *control files* are created, containing names of files to be transferred. When necessary, local files are copied into *data files* for transmission.
- uux(1C)** Execute command. An *execute file* is created, containing a UNIX command to be executed and its arguments. A *control file* is created that includes all files that must be transferred to execute the command, including the *execute file* itself. When necessary, local files are copied into *data files* for transmission. Any output from the command will also be written to *data files*.

The following four commands provide UUCP status information:

- uulog(1C)** Display selected information from the UUCP log.
- uname(1C)** Display the names of all remote hosts that are directly accessible via UUCP.
- uusnap(8C)** Provide a snapshot of the current queue, including the number of work files, data files, and execute files for each site.
- uuq(1C)** A variant of *uusnap*, lists files and *uux* commands queued for each site. *Uuq* also permits the UUCP administrator to delete jobs.

The following four commands provide miscellaneous support services:

- uudecode(1C)** The decoder for files created by *uuencode*, below.
- uuencode(1C)** A filter to convert binary files into printable ASCII. This is useful when transferring object files over communications links that do not support 8-bit transfers.
- uupoll(8C)** A user utility to conveniently fork the UUCP daemon, *uucico*.
- uusend(1C)** A utility to send files to remote sites more than one "hop" distant.

### 2.1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

```
uucp [ option ] ... source ... destination
```

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

- f** Don't make directories when copying the file. The default is to make the necessary directories.
- C** Copy source files to the spool directory. The default is to use the specified source when the actual transfer takes place.
- ggrade** Put *grade* in as the grade in the name of the work file. This is a single character in the range [0-9][A-Z][a-z]. The *grade* will be used by *uucico* to establish the priority of



requests. 0 is the highest (best) grade; z is the lowest (worst). The default *grade* for *uucp* is n.

-m Send mail on completion of the work.

-nuser Notify *user* on the destination system that a file was sent.

The following options are used primarily for dchugging, or when *uucp* is invoked from other programs:

-r Queue the job but do not start *uucico*. The assumption is that *uucico* will be started at a later time, perhaps by *crond*(8) or *uupoll*.

-sdir Use directory *dir* for the top level spool directory.

-xnum *N* is the level of dchugging output desired. This option requires the user to have read permission to the UUCP control file *L.sys*.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "?\*|]"; these and other shell characters such as "!<>" will need to be quoted or escaped. If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

```
uucp *.c usg/usr/dan
```

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the "usg" machine.

The source and/or destination names may also contain a *~user* prefix. This translates to the login directory on the specified system. A lone *~* prefix is expanded to the name of the specified system's public access directory, usually */usr/spool/uucppublic*. For names with partial path-names, the current directory is prepended to the file name. File names with *../* are not permitted.

The command

```
uucp usg!~dan/*.* ~dan
```

will set up the transfer of files whose names end with ".\*" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

- [1] Copy source to destination on local system.
- [2] Receive files from a remote system.
- [3] Send files to a remote system.
- [4] Send files from remote system to another remote system.
- [5] Receive files from remote system when the source pathname contains special shell characters as mentioned above.

After the work has been set up in the spool directories, the UUCP daemon *uucico* is started to try to contact the other machine to execute the work (unless the *-r* option was specified).

#### Type 1

*Uucp* makes a copy of the file. The *-m* option is not honored in this case.

#### Type 2

A one line *work file* is created for each file requested and put in the C. spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

- [1] R

- [2] The full path-name of the source or a `~user/path-name`. The `~user` part will be expanded on the remote system.
- [3] The full path-name of the local destination file. If the `~user` notation is used, it will be immediately expanded to be the login directory for the user.
- [4] The user's login name.
- [5] A '-' followed by an option list.

### Type 3

For each source file, a *work file* is created. A `-C` option on the `uucp` command will cause the *data file* to be copied into the spool directory and the file to be transmitted from the copy; the copy is deleted when the transfer completes. The fields of each entry are given below.

- [1] S
- [2] The full-path name of the source file.
- [3] The full-path name of the destination or `~user/file-name`.
- [4] The user's login name.
- [5] A '-' followed by an option list.
- [6] The full path-name of the local source file. If the `~user` notation is used, it will be immediately expanded to be the login directory for the user. If the `-C` option was used, this will be the name of a *data file* in the spool directory.
- [7] The file mode bits of the source file in octal print format (e.g. 0666).
- [8] The user to notify on the remote system that the transfer has completed.

### Type 4 and Type 5

`Uucp` generates a `uucp` command and sends it to the remote machine; the remote `uucico` executes the `uucp` command.

## 2.2. Uux - UNIX To UNIX Execution

The `uux` command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the `uux` command is

```
uux [-] [option] ... command-string
```

where the *command-string* is made up of one or more arguments. All special shell characters such as "`<>|*?!`" must be quoted either by quoting the entire *command-string* or quoting the character as a separate argument. Within the *command-string*, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a "`!`" will not be treated as files. (They will not be copied to the execution machine.) The '-' is used to indicate that the standard input for *command-string* should be inherited from the standard input of the `uux` command.

The options, used mostly for debugging and by other programs, are:

- aname Use *name* as the requestor of the `uux` command, instead of the real system and login names. Unlike most other UUCP arguments, *name* may consist of a chain of system names separated by '`!`' characters, as in:  

```
uux - -r -aihnp4!decwrl!pyramid!csg seismo!rmail rick
```
- C Copy source files to the spool directory. Same as for `uucp`.
- ggrade Put *grade* in as the grade in the name of the work file. Same as for `uucp`. The default *grade* for `uux` is A.
- n Do not mail an acknowledgement to the requestor of the command. Normally the execution daemon, `uuxqt`, will mail a message to the user who entered the `uux` command. This message includes the status return value that the program exited with. The `-n` option requests that this message not be sent.

- r Do not start the UUCPcp daemons *uucico*(8C) or *uuxqt*(8C) after queuing the job.
- xnum Num is the level of debugging output desired.
- z Mail an acknowledgement to the requestor only if the command fails, that is, the command exits with a non-zero status.

The command

```
pr abc | uux - usg!lpr
```

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

*Uux* generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the X. spool directory for local execution, or in the D.hostnameX directory for transfer to the remote system.

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by *uucico*. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The *execute file* will be processed by the *uuxqt*(8C) program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

#### User Line

```
U user system
```

where the *user* and *system* are the requestor's login name and system.

#### Required File Line

```
F file-name real-name
```

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

#### Standard Input Line

```
I file-name .
```

The standard input is either specified by a '<' in the command-string or inherited from the standard input of the *uux* command if the '-' option is used. If a standard input is not specified, /dev/null is used.

#### Standard Output Line

```
O file-name system-name
```

The standard output is specified by a '>' within the command-string. If a standard output is not specified, /dev/null is used. (Note - the use of ">>" is not implemented.)

#### Status Return Line

```
N
```

Normally *uuxqt* mails an acknowledgement message to the requestor after the command completes. The message includes the status return value that the program exited with. This line inhibits mailing of the acknowledgement message. It is generated by the -n option of *uux*; it is also quietly assumed by *uuxqt* on the command *rmail*.

**Error Status Return Line****Z**

A variant of the *Status Return* line, this line indicates that an acknowledgement should be mailed only if the command's status return is non-zero, i.e., the program exited with an error. This line is generated by the *-z* option of *uux*. It is also quietly assumed by *uuxqt* on the command *rnews*. If both the *Z* and *N* lines appear, the *Z* line has precedence.

**Requestor Line****R requestor**

where *requestor* is a complete return mailing address to the original requestor. This line is generated by the *-a* option of *uux*, and is used to override the mail return address implied by the *User* line. This is commonly used by mailers and programs like *uuseed* that know how to "hop" a file from system to system.

**Command Line****C command [ arguments ] ...**

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file (usually */bin/sh*). In addition, a shell "PATH" statement is prepended to the command line.

After execution, the temporary standard output file is copied to or set up to be sent to the proper place.

**3. SYSTEM AND ADMINISTRATIVE UTILITIES**

UUCP includes four system utilities; these are not normally referenced by users. All except *uucpd* reside in the UUCP administrative directory, */usr/lib/uucp*. These include:

- uucico*(8C)      Copy In, Copy Out. This is the primary UUCP daemon.
- uuclean*(8C)    A handy utility to clean up the UUCP spool directories.
- uucpd*            The UUCP TCP/IP daemon. This daemon "answers" the connection request from a remote *uucico* to a TCP/IP socket. It is functionally a stripped-down version of *rlogind*(8) that provides full 8-bit communication. (Note: this utility does not have a *man* page.)
- uuxqt*(8C)       Execution Daemon. This is forked by *uucico* to interpret execution files transferred from a remote system.

**3.1. Uucico - Copy In, Copy Out (UUCP Daemon)**

*Uucico* is the "heart" of the UUCP system. The program performs the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways;

- a) by a system daemon (such as *cron*(8)),
- b) by one of the *uucp*, *uux*, *uuxqt* or *uupoll* programs,
- c) directly by the user (this is usually for testing),

- d) by a remote system. (The *uucico* program should be specified as the "shell" field in the */etc/passwd* file for the UUCP logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (*-s* option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

- ggrade* Set the minimum grade of this *uucico* run to *grade*. Only files of this grade or better will be transferred.
- rl* Start the program in *MASTER* mode. This is used when *uucico* is started by a program or *cron* shell.
- ssys* Do work only for system *sys*. If *-s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

- ddir* Use directory *dir* for the top level spool directory.
- xnum* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

#### Scan For Work

The names of the work related files in a spool subdirectory have format

type . system-name grade number

where:

- Type* is an upper case letter, ( *C* - work (copy command) file, *D* - data file, *X* - execute file);
- System-name* is the remote system;
- Grade* is a character in the range [0-9][A-Z][a-z];
- Number* is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the appropriate spool directory for *work files* (files with prefix *C.*). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

#### Call Remote System

The call is made using information from the *control* files that reside in the */usr/lib/uucp* directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* control file. The information contained for each system is;

- [1] system name,
- [2] times to call the system (days-of-week and times-of-day),
- [3] the *caller*, that is, the type of device to be used for the call,

- [4] the line speed or network number (as appropriate),
- [5] telephone number or device name (as appropriate),
- [6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

#### Line Protocol Selection

The remote system sends a message

*Pproto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the *proto-list* for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

*Ucode*

where *code* is either a one character protocol letter or N which means there is no common protocol.

The following protocols are implemented in 4.3BSD UUCP:

- g** General. Default for dialup or hardwired lines, supported by all versions of UUCP. This protocol employs small (64 byte) data packets with checksums and packet-by-packet retransmission. This ensures reliable and efficient transfers over slow and noisy links like 1200-baud dial-up lines. These same characteristics make the *g* protocol bulky and slow over error free links, and very expensive on public data-switched networks.
- f** Optimized for use on X.25 PAD public data-switched networks. The protocol employs larger (256 byte) data packets, passes no control characters except CR, and uses only a 7-bit data path. (Note that the files transferred may still contain full 8-bit data.) It assumes that the link is "mostly" error-free, calculating a checksum for the entire file only. When an error is detected, the entire file is retransmitted.
- t** Optimized for use on TCP/IP networks and other completely error free links. It employs large (1024 byte) packets, and uses the full 8-bit data path.

Note: AT&T System VR2 UUCP supports the *x* (*X.25*) and *e* (*Error Free*) protocols, which provide functionality similar to the 4.3BSD *f* and *t* protocols, respectively. They are incompatible, however. Thus when attempting to connect two systems via X.25 or an local area network, it is not adequate for both systems to simply "support X.25" or "support error free transfers." Both must support the same UUCP protocols.

#### Work Processing

The initial roles ( *MASTER* or *SLAVE* ) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the *-r1 uucico* option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

- S send a file,
- R receive a file,
- C copy complete,
- X execute a *uucp* command, and
- H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, *YN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory using *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the *TM*. spool subdirectory.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of its spool directory. If work for the *MASTER*'s system exists in the *SLAVE*'s spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

#### Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *-s* option was specified.

### 3.2. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the *X*. spool directory for *execute files*. Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the *uux* section above.

#### Command Execution

The execution is accomplished by executing a *sh -c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

### 3.3. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the *cron(8)* daemon, once a day. Its function is to remove files from the spool directories which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

- ddir*     The directory to be scanned is *dir*.
- m*        Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the *uucp* programs since the *setuid* bit will be set on these programs. The mail will therefore most often go to the owner of the *uucp* programs.)

- nhours* Change the aging time from 72 hours to *hours* hours.
- ppre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
- xnum* This is the level of debugging output desired.

#### 4. SYSTEM CONTROL FILES

Seven *Control Files* are referenced by the UUCP utilities. All live in the UUCP administrative directory, */usr/lib/uucp*. These are ASCII files, and can be modified using standard text editors such as *vi* and *ex*. Lines beginning with a '#' character are comments; lines ending with a '\' are continued on the next input line.

- L-devices(5) Declares all devices that are available to *uucico* for calling out.
- L-dialcodes(5) Phone number prefixes. Used to map alphabetic prefixes on phone numbers from *L.sys* to real phone numbers. Also useful to keep a phone number database outside of *L.sys*.
- L.sys(5) Systems. Declares all "adjacent" UUCP hosts, with directions on how to reach them.
- L.aliases(5) Contains aliases used to map obsolete or truncated host names to the correct names.
- L.cmds(5) Commands Permissions. Declares those commands for which remote *uux* execution is permitted.
- SQFILE Sequence-number check file. (Optional)
- USERFILE(5) Directory Tree Permissions. Specifies the set of directory trees that a particular user or host may reference.

A general description of each file follows; see the *man* pages for complete information. Examples of the six standard files are included in the distribution in the */usr/lib/uucp/UUAIDS* directory.

##### 4.1. L-devices – UUCP Devices File

This file declares all devices that are available to *uucico* for calling out. The special device files are assumed to be in the */dev* directory. The format for each entry is

caller line call-unit class dialer [chat....]

where;

- caller is the caller mechanism, that is, the type of device to be used. This can be one of ACU (for Automatic Call Units (modem)), DIR (direct hardwired), PAD (X.25/PAD), and others.
- line is the device for the link. For example, *cul0* for a modem, *tty10* for a hardwired line.
- call-unit is the automatic call unit associated with *device*. This is used on autodialers such as the Racal-Vadic MACS and the DEC DN-11 that use one device for data, and a second device for dialing. If unused, this field must contain a placeholder such as "unused" or "0". Some modems use this field to specify tone or pulse dialing.
- class is the line speed, plus an optional alphabetic prefix. The prefix can be used to distinguish among different devices that have identical *caller* and line speed.
- dialer applies to ACU devices only; this is the type or brand name of the modem. Supported modems include DN11 (DEC DN-11), hayes (Hayes Smartmodem), vadic (Racal-Vadic 3451), ventel (VenTel 212A), and others.
- chat refers to an *expect/send* script, similar to that provided in *L.sys*. The difference is that the script in *L-devices* is executed before the connection is established, while the script in *L.sys* is executed afterwards.



The line

```
ACU tty47 unused 1200 hayes
```

would be set up for a system which had device `tty47` wired to a Hayes "Smartmodem 1200" for use at 1200 baud.

#### 4.2. `L-dialcodes` – Phone Number Prefix File

This file contains entries with location abbreviations used in the `L.sys` file (e.g. `py`, `mh`, `boston`). The entry format is

```
abb dial-seq
```

where;

`abb` is the abbreviation,

`dial-seq` is the dial sequence to call that location.

The line

```
py 165-
```

would be set up so that entry `py7777` would send 165-7777 to the dial-unit.

#### 4.3. `L.aliases` – Hostname Aliases File

This file defines mapping (aliasing) of remote host names. This is intended for compensating for systems that have changed names, or do not provide their entire machine name (like most USG systems). It is also useful when a machine's name is not obvious or commonly misspelled.

Each line is of the form

```
real-name alias-name
```

where *real-name* is the full, correct name for the host, and *alias-name* is the old or truncated name.

#### 4.4. `L.sys` – UUCP Systems File

Each entry in this file represents one system which can be called by the local uucp programs. The format for each entry is

```
system times caller class device/phone-number [login]
```

where;

`system` is the hostname of the remote system.

`times` is a keyword-encoded string that indicates the days-of-the-week and times-of-day when the system may be called. For example `MoTuTh0800-1730` would denote Monday, Tuesday, and Thursday, between 8 a.m. and 5:30p.m.

The day portion may be a list containing any of `Su`, `Mo`, `Tu`, `We`, `Th`, `Fr`, `Sa`, or `Wk` for any week-day or `Any` for any day.

The time should be a range of times (as in `0800-1230`). If no time portion is specified, any time of day is assumed to be acceptable for the call.

`caller` is one of the caller device-types listed in *L-devices*.

`class` is the line speed for the call (e.g., 300, 1200, 9600), plus an optional alphabetic prefix. Network devices use this field for the network port number.

`phone` is the the phone number to call (for ACU devices) or the device filename. A phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. `mh5900`, `boston995-9980`).

`login` is a script describing how to log in to the remote host. It is expressed as a series of fields and subfields in the format

expect send [ expect send ] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received. The *send* string is normally terminated with carriage-return; an empty *send* string will send only a carriage-return.

The expect field may be made up of subfields of the form

expect[-send-expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

A typical entry in the L.sys file would be

```
sys Any ACU 1200 mh7654 login:--login: uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

#### 4.5. L.cmds - Commands Permissions File

This file contains a list of commands, one per line, that are permitted for remote execution via *uux*. This list should be chosen with great care, since commands that take filenames as arguments will allow users to easily circumvent UUCP's security. For most sites, *L.cmds* should only include the lines:

```
rmail
ruusend
```

#### 4.6. SQFILE - Sequence Check File (Optional)

This file contains an entry for each remote system with which this site agrees to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, which could indicate that an unauthorized connection has been attempted, the entry will have to be adjusted.

This facility is technically no longer supported in 4.3BSD UUCP, since it was hardly ever used and consumed precious memory space on PDP-11 systems. The compile-time `#define GNXSEQ` can be set to enable sequence checking should it be needed.

#### 4.7. USERFILE - Pathnames Permissions File

This file contains user accessibility information. It specifies four types of constraint;

- [1] which files can be accessed by a normal user of the local machine,
- [2] which files can be accessed from a remote computer,
- [3] which login name is used by a particular remote computer,
- [4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

```
login,sys [ c ] path-name [ path-name ] ...
```

where;

login        is the login name for a user or the remote computer,  
 sys         is the system name for a remote computer,  
 c           is the optional *call-back required* flag,  
 path-name   is a path-name prefix that is acceptable for user.

The constraints are implemented as follows.

- [1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.
- [2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.
- [3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name. Note: This constraint, although stated in the original Nowitz UUCP document, was not implemented in Version 7 UUCP. For all practical purposes, a remote computer's login was not validated by UUCP. This is still the case in 4.3BSD. Remote login checking *is* implemented in AT&T's System VR2.2 release, and in the UUCP provided with Digital Equipment Corporation's ULTRIX. HoneyDanBer analogously requires all remote logins to be listed in its *Permissions* file.
- [4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

```
dan, /usr/dan
```

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines

```
root, /
, /usr
```

allows any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file.

## 5. SPOOL FILES

Spool Files contain UUCP transfer requests and data. Most have been described in detail earlier in this document.

All spool files live in the */usr/spool/uucp* directory tree. To keep the spool directory from becoming hopelessly cluttered, each type of spool file is kept in its own subdirectory. The name of the directory is the same as the common prefix of the filename. For example, *work files* (files beginning with C.) are kept in the C. directory; *execute files* (which begin with X.) are kept in the X. directory, and so on.

A total of ten spool subdirectories are used, one of which is optional:

- |         |  |
|---------|--|
| C.      | <i>Work files.</i>   |
| CORRUPT | Corrupted <i>work</i> and <i>execute</i> files. <i>Uucico</i> and <i>uuxqt</i> will deposit C. and X. files here when they are unable to parse them. A notice will also be placed in the |

	UUCP log.
D.	Data files received from remote hosts.
D.hostname	Data files to be sent to remote hosts.
D.hostnameX	Execution files to be sent to remote hosts.
LCK	Per-device and per-site lock (LCK.) files. (Optional)
STST	Per-site system status files.
TM.	Temporary files used in data transfer. When the transfer is complete, the file is typically <i>mv</i> 'ed to the D. or X. directory.
X.	Execution files received from remote sites.
XTMP	Temporary files and home directory for <i>uuxqt</i> .

The following sections describe only those spool files that were not discussed earlier.

### 5.1. LCK – lock files

Lock files are created for each device in use (except for TCP/IP sockets) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..*str*

where *str* is either a device or system name. The files may be left in the spool directory if *uucico* aborts. They will be ignored (reused) after 90 minutes. When runs abort and calls are desired before the time limit expires, the lock files should be removed. If the LCK. subdirectory is used, its access mode can be set to 777, thus allowing normal users to remove dead lock files when necessary.

### 5.2. STST – system status files

These files are created in the STST subdirectory by *uucico*. They contain information of failures such as login, dialup, or sequence check, and will contain a *TALKING*, *RECEIVING*, or *SENDING* status when two machines are conversing. The file name is STST/*system*, where *system* is the host name of the remote machine.

For ordinary failures (dialup, login), the file indicates the time of the last failure; this allows *uucico* to avoid retrying the failed call too soon. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted. The easiest way to do this is to use the *uupoll* command to force *uucico* to start up.

### 5.3. TM – temporary data files

These files are created in the /usr/spool/uucp/TM. directory while files are being copied from a remote machine. Their names have the form

TM.*pid*.*ddd*

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received. After the entire remote file is received, the TM file is moved to the requested destination, often the X. or D. directory. If processing is abnormally terminated or the move fails, the file will remain in the TM. directory.

The stranded files should be periodically removed; the *uuclean* program is useful in this regard. The command

uuclean -d/usr/spool/uucp/TM. -pTM.

will remove all TM files older than three days.

## 6. LOG FILES

The following files provide a history of UUCP activities. All live in the spool directory, `/usr/spool/uucp`. They grow forever, and must be periodically trimmed or deleted; this is usually done weekly (or daily) via *cron*.

- |         |   |
|---------|---|
| AUDIT   | This is a directory of audit trail files, one file per site. <i>Uucico</i> uses an audit file for debugging output whenever it is run with <i>debug</i> enabled (via the <code>-x</code> option or a <code>SIGFPE</code> signal), but the standard message output file <code>stderr</code> is not available.  |
| ERRLOG  | This is an oft-forgotten log of UUCP "Assert" errors. An Assert error is a catastrophic and unrecoverable failure of the UUCP system. These include spool directories or control files that cannot be opened, an unexpected error return from a system call, or an "impossible case" in a utility's control flow.<br><br>Utilities that abort with an Assert error return a status code of -1. If a user reports <i>uucp</i> or <i>uux</i> dying with a message like "uux failed, status -1," then the ERRLOG file should be checked. |
| LOGFILE | This is the primary UUCP log. All UUCP activity is recorded here, including queue requests from <i>uucp</i> and <i>uux</i> , attempted connections, file transfers, and communications failures.  |
| SYSLOG  | This is a log of file transfer statistics: number of bytes, time required, and number of packet retries. The effective data rate can be calculated simply by dividing the number of bytes by the time; low data rates or a large number of retries implies that the communication link may be marginal.   |

Optionally, one *LOGFILE* per site may be maintained in the *LOG* subdirectory. This option can be selected at UUCP compile time via the `LOGBSITE` #define in `uucp.h`.

## 7. ADMINISTRATION AND SYSTEM SECURITY

### 7.1. UNIX System Files

#### `/etc/passwd`

UUCP requires a login in `/etc/passwd`; at its simplest the entry would be

```
uucp::66:1:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

This user should own all the UUCP files and utilities. Remote sites wishing to call in for UUCP transfers would login to *uucp* (with the correct password, if any), and get *uucico* as their "shell." Since *uucico* would be called without any options, it would run in *SLAVE* mode, thus responding correctly to the remote system, which would be in *MASTER* mode.

The directory `/usr/spool/uucppublic` should be created with 777 access modes, owned by *uucp*. In addition to serving as the home directory for UUCP remote logins, *uucppublic* provides a "public-access" directory where any user can read, write, or transfer files.

There are a number of security problems with using a single login, not the least of which is that superuser permission would be necessary to edit the *control* files. A better arrangement would be:

```
uucp::66:1:UUCP Administrator:/usr/lib/uucp:
```

```
nuucp::67:1:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

This provides one login for the UUCP administrator (which must be kept secure!) and a second for remote machines to use for login. A still more elaborate setup would use a separate login for each remote site, and possibly provide the administrator with a choice of shells:

```

uucp::66:1:UUCP Administrator:/usr/lib/uucp:
UUCP::66:1:UUCP Administrator:/usr/lib/uucp/bin/csh
Uhosta::6001:1:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
Uhostb::6002:1:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
Uhostc::6003:1:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico

```

It is assumed that the login name used by a remote computer to dial in is not the same as the login name of a normal user of the machine. However, several remote computers may employ the same login name.

Note that `uucppublic` is *not* used as the home directory for `uucp` when it logs into a regular shell. This would be an extreme security hazard, since anyone could slip a "Trojan horse" into a `.profile` or `.cshrc` file, which would be automatically executed when the UUCP administrator logged in.

`/etc/rc`

The system startup file, `/etc/rc`, should clean up any stray lock files with the line

```
rm -f /usr/spool/uucp/LCK.*
```

or, if the LCK subdirectory is being used,

```
rm -f /usr/spool/uucp/LCK/LCK.*
```

`/etc/services`

If UUCP is to be used over TCP/IP links, then an entry for UUCP's port number should be added to `/etc/services`:

```
uucp 540/tcp uucpd # UUCP TCP/IP
```

## 7.2. Shell Scripts For Periodic Cleanup

The UUCP system has a fairly large number of activities that must occur periodically. These include starting `uucico` to process queued requests, running `uuclean` to remove old spool files, and shuffling the boundlessly-growing log files. Some sites will also want to poll other sites periodically.

While it's possible to put all the necessary commands into `cron`'s control file `/usr/lib/crontab`, this would be extremely awkward. The usual technique is to use three separate shell scripts, one each for hourly, daily, and weekly operations. Examples are provided in the `UUAIDS` directory; the following sections provide some specific recommendations.

### Hourly

Activities that should occur hourly include:

- Polling of selected sites. Sites that have no dial-out capability will need to be periodically polled. The `uupoll` command works well for this.
- Start `uucico` to complete all unfinished work. This can be as simple as:
 

```
uucico -r1 &
```

### Daily

The daily script should be started by `cron` in the wee hours, around 4 a.m. Activities that should occur daily include:

- Call `uuclean` to remove old spool files. The preferred technique is something like the following:

```

cd /usr/lib/uucp
deadtime='expr 24 7'
uuclean -d/usr/spool/uucp/AUDIT -n72
uuclean -d/usr/spool/uucp/LCK -pLCK. -pLTMP. -n24
uuclean -d/usr/spool/uucp/STST -n72
uuclean -d/usr/spool/uucp/TM. -pTM. -n72
uuclean -d/usr/spool/uucp/XTMP -n72
uuclean -d/usr/spool/uucp/X. -pX. -n$dcadtime
uuclean -d/usr/spool/uucp/C. -pC. -n$dcadtime
uuclean -d/usr/spool/uucp/D. -pD. -n$dcadtime
uuclean -d/usr/spool/uucp/D.'uname -l' -pD. -n$dcadtime
uuclean -d/usr/spool/uucp/D.'uname -l'X -pD. -n$dcadtime

```

In this example, Audit files, Lock files, System Status files, tmp files, and *uuxqt* output files are cleaned up every 72 hours (3 days). (LTMP. files are temporary files created by the lock mechanism; they are rarely around for more than a few seconds. Note, the above assumes that the LCK subdirectory is being used.) All normal data files are cleaned up every  $24 * 7$  hours, or every 7 days.

- Shuffle the log files. At the very least, LOGFILE should be moved to LOGFILE.old, and SYSLOG moved to SYSLOG.old. Busy sites may want to use *compress(1)* to squeeze down the old files.
- Use *find(1)* to clean up the /usr/spool/uucppublic directory. If left unattended, garbage will gradually accumulate there until it fills the file system.

### Weekly

Small sites with very little traffic may choose to shuffle the log files once per week, instead of once per day. The weekly script should, like the daily script, be run early in the morning.

### 7.3. Connecting new systems

When first connecting a new machine to a UUCP network, it is useful to try and establish a connection with *tip* or *cu* first. The UUCP administrator will quickly become aware of any special facilities that are going to be required, for example: What lines and modems are to be used? Is the connection through different hardware and carriers? Does the remote system care about parity? What speed lines are being used and do they cycle through several speeds? Is there a line switch front end that will require special login dialogue in *L.sys*?

Once a successful login is achieved "by hand," the administrator should have enough information to allow the correct setup of the *control* files in

The UUCP administrator should then negotiate with the remote site's UUCP administrator as to who (if anyone) will do polling and when. Both administrators must set up the relevant accounts and passwords. The local administrator should decide on what permissions and security precautions are to be observed. Testing time and facilities will need to be arranged to complete initial connection testing between the systems.

### 7.4. Miscellaneous Security Issues

The UUCP system, left unrestricted, will let any outside user execute any commands and copy any files that are accessible to the *uucp* login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the UUCP system.

- The login for *uucp* does not get a standard shell. Instead, *uucico* is started. Therefore, the only work that can be done is through *uucico*.

- A path check is done on file names that are to be sent or received. *USERFILE* supplies the information for these checks. *USERFILE* can also be set up to require call-back for certain login-ids. (See the description of *USERFILE* above.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- *Uuxqt* is restricted via the *Lcmds* file to a small list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the *Lcmds* file. The administrator may modify the list or remove the restrictions as desired.
- All the utilities except *uudecode*, *uuencode*, and *uusend* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard shell with a *uucp* login. Optionally, the utilities may belong to group *daemon* and be given "setgid" permissions (mode 06111). *Uuxqt* should only permit other UUCP programs to execute it; its mode should be 04100 or 06110.
- The *control* files *L.sys*, *USERFILE*, and *SQFILE* contain highly sensitive information. They should be owned by the *uucp* login, with read and write permission granted only to the owner (mode 0600).

## 8. INSTALLING THE UUCP SYSTEM

The source for the UUCP system resides in the */usr/src/usr.bin/uucp* directory. The *README* file includes complete instructions on how to rebuild the UUCP system from source.

For most environments, only two files will need to be modified: *uucp.h* includes a large number of tunable system-dependent parameters, including operating system type, devices to be supported, and a variety of optional features. The *Makefile* may also have to be modified, particularly if you chose to keep certain files in different directories from usual.

## 9. ACKNOWLEDGEMENTS

4.3BSD UUCP was a group development effort, involving the contributed work of over one hundred members of the USENET community. We're extremely grateful to them all.

Special thanks go to the following individuals, whose contributions were especially valuable:

- Rick Adams (Center for Seismic Studies) coordinated the 4.3BSD UUCP release, incorporating (and often correcting) hundreds of bug fixes that were posted on the USENET and mailed to him directly. Rick also managed to find time to add many enhancements and corrections of his own.
- Tom Truscott (Research Triangle Institute) and Bob Gray (then with PAR Tech Corp, now at Univ of Colorado) coordinated the 4.2BSD UUCP release, which was also a group effort. Tom has continued to provide enhancements and fixes in 4.3BSD.
- Guy Harris (then with Computer Consoles, Inc., now with Sun Microsystems) contributed many general bug fixes; in particular, he was the first to isolate the infamous 4.2BSD "TIMEOUT" bug.
- Lou Salkind (New York University) wrote the *uuq* utility.
- James Bloom (U.C. Berkeley) isolated a major day-one bug in the g-protocol driver that had eluded many people's attempts to squash it.
- Piet Beertema (Centrum voor Wiskunde en Informatica, Amsterdam) wrote the f-protocol to support "mostly error-free links"; Robert Elz (University of Melbourne) modified the protocol specifically for X.25/PAD.
- Peter Honeyman (Princeton) assisted Rick by providing information on the facilities provided in HoneyDanBer UUCP; Rick then added many HDB-compatibility features and HDB-like extensions to 4.3BSD UUCP.
- Ross Green (U.C. Berkeley) produced the first revision of this chapter, updating the aging Nowitz document to more closely reflect reality.

Thanks again to everyone who contributed. Berkeley UUCP continues to be a product of its own users, and would not exist as it does today without them.



## USENET Version B Installation

Matt Glickman  
Computer Science Division  
Department of Electrical Engineering and Computer Sciences  
University of California  
Berkeley, California 94720

Revised by Mark Horton for version 2.10

Revised by Rick Adams for version 2.10.3

### 1. Introduction

This document is intended to help a USENET site install and maintain the network news software. Please ask questions of Rick Adams†; such questions will help to point out areas that need to be addressed here.

The overall order of things to do is:

- (a) Find somebody to link up with. You need a network connection of some kind, for example, ARPANET or UUCP. If you must use UUCP and have no connections, you must have at least a dialup and preferably a dialer, and find someone willing to call your machine. The USENET directory may be helpful in finding some other site geographically near yours to hook up to.
- (b) Create a *localize.sh* script to make local changes to the makefile and *defs.h* files. (Section 2 gives more details about creating *localize.sh*.) Once you're finished editing *localize.sh*, create a *defs.h* and *Makefile* tailored for your site with the command  
sh *localize.sh*  
Inspect *defs.h* and *Makefile* to ensure that all your local customizations got into your final versions. If you saw a "?" when you ran *localize.sh*, one or both of the files is certainly wrong. It's a good idea to anchor the patterns in *localize.sh*'s *ed(1)* scripts, especially in its *Makefile*-editing lines. For instance, use */UUXFLAGS/* instead of */UUXFLAGS/*.
- (c) Compile the software using the *make(1)* command.
- (d) *Su(1)* and type "make install". This will copy the files out to the right place and make directories containing most of the important files. It will configure you in with a connection to *oopsvax* via UUCP links. This is undoubtedly wrong, so you will have to configure links as needed. If you are upgrading from a version older than 2.10.3, do "make update". This will cause various checks to be performed on important files in *LIBDIR*. The results will be reported to you. If you are not sure if you should do "make update", do it. It will not hurt anything if you have already done it.
- (e) After editing the configuration table, get your contact at the other end of the link to add you to their netnews *sys* file.
- (f) Post a message to the *to.sysname* newsgroup which should be set up to go only to the site you are linked to, as a test. Have the other person send a message to your system using the same

† ARPANET: rick@seismo.CSS.GOV, UUCP: seismo!rick



mechanism. If this doesn't work, find the problem and fix it. (Please don't use `net.test` unless there is no alternative. It is almost always possible to use `test`, or `to.sysname` or some `local.test` group, instead of `net.test`.)

- (g) Fill out a USENET directory form (the file `dirform` in the `misc` directory). Post a copy to the USENET newsgroup `net.news.newsite` and mail a copy to `cbsgd/uucpmap`.
- (h) Format the document "*How to Read the Network News*" (the file `howto.mn` in the `doc` directory), the document "*How to Use USENET Effectively*" (the file `manner.mn` in the `doc` directory) and the document "*Copyright Law*" (the file `copyright.mn` in the `doc` directory) and post them to your general newsgroup with a long expiration date. You can use `inews(1)` or `postnews(1)` to do this.
- (i) It will probably be necessary to fix your `uucp` commands to allow `rnews` and to support the `-z` and `-n` options (if you are lucky enough to have the source).

## 2. Installation

### 2.1. Configuration

Local configuration of the USENET version B software requires you to edit a few files. Most importantly, the `defs.h` and `Makefile` files must be created from their templates `defs.dist` and `Makefile.dist`. You should create a shell script called `localize.sh` which copies the files and makes local changes to the copies. Even for a completely vanilla site, some changes will be necessary. For example, your script should start with `localize.v7` or `localize.usg`. You should include the name of the local organization (MYORG) and the uid of the local news super user (ROOTID). You should also choose how your hostname will be determined. If you are a USG site, define `UNAME` in `defs.h`. If you are running 4.[23] BSD, define `GNAME` in `defs.h`. If you have your UUCP name in `/etc/uucpname`, define `UUNAME` in `defs.h`. Otherwise, news will look in the file `/usr/include/whoami.h` for a line of the form

```
#define sysname your-sysname
```

If you are running System 3 or System 5, you are a USG site. Otherwise, unless you are in AT&T, you are probably a V7 site. The previously mentioned defines are the only modifications that are *necessary* to install news at your site. However, you will probably want to change some of the ones listed below. If your compiler does not accept "(void)", the simplest thing to do is add "`-Dvoid=int`" to the `CFLAGS` line in the `Makefile`.

A sample `localize` shell script can be found in `localize.sample`. The most important parameters are:

#### 2.1.1. ROOTID

The numerical uid of the person who is the news super user. This should not be set to 0. Normally it is set to the uid of the news contact person for the site. If it is not defined, the uid of `NOTIFY` will be looked up in `/etc/passwd` and used instead.

#### 2.1.2. N\_UMASK

Mask for `umask(2)` system call. Set it to something like 022 for a secure system. Unsecure systems might want 002 or 000. This mask controls the mode of news files created by the software. Insecure modes would allow people to edit the files directly.

#### 2.1.3. DFLTEXP

The default number of seconds after which an article will expire. Two weeks (1,209,600 seconds) is the default choice. If you wish to expire articles faster than two weeks, it is recommended that you use the `-e` flag to expire instead of decreasing `DFLTEXP`.

#### 2.1.4. HISTEXP

Articles which were posted more than **HISTEXP** ago are considered too old and are moved into the junk directory. This is because they are too old to be in the history file, so it is impossible to tell if they really should be accepted or are endlessly looping around the network. (This was theoretically possible before this feature was added.) The articles are removed after **DFLTEXP** seconds, but a copy of their "Message-ID" is kept in the history file for **HISTEXP** seconds (the default is 4 weeks).

#### 2.1.5. DFLTSUB

The default subscription list. If a user does not specify any list of newsgroups, this will be used. Popular choices are **all** and **general**, **all,general**.

#### 2.1.6. TMAIL

This is the version of the Berkeley *Mail*(1) program that has the **-T** option. If left undefined, the **-M** option to *readnews*(1) will be disabled.

#### 2.1.7. ADMSUB

This newsgroup (or newsgroup list) will always be selected unless the user specifies a newsgroup list that doesn't include **ADMSUB** on the command line. That is, as long as the user doesn't use the **-n** flag to *readnews* on the command line, **ADMSUB** will always be selected. This is usually set to **general**. (The intent of this parameter is to have certain newsgroups which users are required to subscribe to. A typical site might require **general**.)

#### 2.1.8. PAGE

The default program to which articles should be piped for paging. This can be disabled or changed by the environment variable **PAGER**. If you have it, the Berkeley *more*(1) command should be used, since the **+** option allows the headers to be skipped.

#### 2.1.9. NOTIFY

If defined, this character string will be used as a user name to send mail to in the event of certain control messages of interest. (Currently these are **newgroup**, **rmgroup**, **sendsys**, **checkgroups**, and **senduname**.) As distributed, mail will be sent to user *usenet*. It is recommended you create such a mailbox (have it forwarded to yourself) if possible, since this makes it easier for another site to contact the site administrator for your site. If you are unable to do this (e.g., you are not the super user) you should change this name to yourself. Also, messages about missing or extra newsgroups are mailed to this user by the **checkgroups** control message.

#### 2.1.10. DFTXMIT

This is the default command to use to transmit news if no explicit command is given in the fourth field of the *sys* file. It normally includes *uux*(1) with the **-z** option. You should install this modification to *UUCP* at once; otherwise your users will start being bombarded with annoying *uux* completion messages. However, you can turn this off to get news installed.

#### 2.1.11. UXMIT

This is the default command used if the **U** flag is present in the flags portion of a *sys* file line. In this case, the second "%s" refers to the name of a file in the news spool area, not a temporary file. It can usually only be used when local modifications are made to the *uucp* system, such as the **-c** option to *uux*.





#### 2.1.12. DFTEDITOR

This is the full path name of the default editor to use during followups and replies. It should be set to the most popular text editor on your system. As distributed, *vi(1)* is used.

#### 2.1.13. UUPROG

If this is defined, it will be used as a command to run when the *senduuname* control message is sent around. Otherwise the command *uuname(1)* will be run. Normally, this program should be placed in *LIBDIR*.

#### 2.1.14. MANUALLY

If this is defined, incoming *rmgroup* messages will not automatically remove the group. News will instead mail a message to *NOTIFY* advising that the group should be removed. If you define *MANUALLY*, you should have *NOTIFY* defined. *MANUALLY* is defined by default to protect you against accidental or malicious removal of an important newsgroup.

#### 2.1.15. NONEWGROUPS

If this is defined, incoming *newgroup* messages will not automatically create the group. News will instead mail a message to *NOTIFY* advising that the group should be created. If you define *NONEWGROUPS*, you should have *NOTIFY* defined. *NONEWGROUPS* is undefined by default to make it easier to automatically maintain the news system.

#### 2.1.16. BATCH

If set, this is the name of a program that will be used to unpack batched articles (those beginning with the character "#"). Batched articles normally are files reading

```
#! news 1234
article containing 1234 characters
#! news 4321
article containing 4321 characters
...
```

Batching is *strongly* recommended for increased efficiency on both sides.

#### 2.1.17. LOCALNAME

Most systems have a full name database on line somewhere, showing for each user what their full name is. Most often this is in the *gecos* field of */etc/passwd*. If your system has such a database, *LOCALNAME* should be left undefined. If not, define *LOCALNAME*, and articles posted will only receive full names from local user information specified in *NAME* or *\$HOME/.name* by the user. If you have a nonstandard *gecos* format (not *finger(1)* or *RJE*) it will be necessary to make local changes to *fullname.c* as appropriate on your system.

#### 2.1.18. INTERNET

If your system has a mailer that understands ARPA Internet syntax addresses ("user@site.domain") turn this on, and replies will use the "From" or "Reply-To" headers. Otherwise, leave it disabled and replies will use the "Path" header.

#### 2.1.19. MYDOMAIN

When generating internet addresses, this domain will be appended to the local site name to form mailing address domains. For example, on system *ucbvax* with user *root*, if *MYDOMAIN* is set to ".UUCP", addresses generated will read "root@ucbvax.UUCP". If *MYDOMAIN* is ".Berkeley.EDU", the address would be "root@ucbvax.Berkeley.EDU". If your site is in more than one domain, use your primary domain. The domain always begins with a period, unless the local site

name contains the domain; in this case **MYDOMAIN** should be the null string.

#### 2.1.20. CHEAP

Do not *chown*(1) spool files to *news*. This will cause the owner of the file to be the person that started the *inews* process. This is used for obscure accounting reasons on some systems.

#### 2.1.21. OLD

Define this if any of your USENET neighbors run 2.9 or earlier versions of B news. It will cause all headers written to contain two extra lines, "Article-I.D." and "Posted", for downward compatibility. Once all your neighbors have converted, you can save disk space and transmission costs by turning this off. It is strongly encouraged that they convert. 2.10.3 is *much* faster than 2.9. The performance difference is dramatic.

#### 2.1.22. UNAME

Define this if the *uname*(2) system call is available locally, even though you are not a USG system. USG systems always have *uname*(2) available and ignore this setting.

#### 2.1.23. GHNAME

Define this if the 4.[23] BSD *gethostname*(2) system call is available. If neither **UNAME** or **GHNAME** is defined, *inews* will determine the name of the local system by reading */usr/include/whoami.h*.

#### 2.1.24. UUNAME

Define this if you keep your UUCP name in */etc/uucpname*.

#### 2.1.25. V7MAIL

Define this if your system uses V7 mail conventions. The V7 mail convention is that a mailbox contains several messages concatenated, each message beginning with a line reading "From *user date*" and ending in a blank line. If this is defined, articles saved will have these lines added so that mail can be used to look at saved news.

#### 2.1.26. SORTACTIVE

Define this if you want the news groups presented in the order of each person's *.newsrc*(5) instead of the active file.

#### 2.1.27. ZAPNOTES

Define this if you want old style notesfile id's in the body of the article to be converted into "Nf-Id" fields in the header.

#### 2.1.28. DIGPAGE

If this is defined, *vnews*(1) will attempt to process the subarticles of a digest instead of treating the article as one big file.

#### 2.1.29. DOXREFS

Define this if you are using *rn*(1). *Rn* uses this option to keep from showing the same article twice.



**2.1.30. MULTICAST**

If your transport mechanism supports multi-casting of messages, define this. Currently ACSNET is the only network that can handle this.

**2.1.31. BSD4\_2**

Define this if you are running 4.2 or 4.3 BSD UNIX†.

**2.1.32. BSD4\_1C**

Define this if you are running 4.1C BSD UNIX.

**2.1.33. SENDMAIL**

Use this program instead of *tecmail*(8) for sending mail.

**2.1.34. MMDF**

Use MMDF instead of *reemil* for sending mail.

**2.1.35. MYORG**

This should be set to the name of your organization. Please keep the name short, because it will be printed, along with the electronic address and full name of the author of each message. Forty characters is probably a good upper bound on the length. If the city and state or country of your organization are not obvious, please try to include them. If the organization name begins with a "/", it will be taken as the name of a file. The first line in that file will be used as the organization. This permits the same binary to be used on many different machines. A good file name would be */usr/lib/news/organization*. For example, an organization might read "AT&T Bell Labs, Murray Hill", "U.C. Berkeley", "MIT", or "Computer Corp. of America, Cambridge, Mass".

**2.1.36. HIDDENNET**

If you want all your news to look like it came from a single machine instead of from every machine on your local network, define **HIDDENNET** to be the name of the machine you wish to pretend to be. Make sure that you have your own machine defined as *ME* in the sysfile or you may get some unnecessary article retransmission.

**2.1.37. NICENESS**

If **NICENESS** is defined, *rnews* does a *nice*(2) to priority **NICENESS** before processing news.

**2.1.38. FASCIST**

If this is defined, *inews* checks to see if the posting user is allowed to post to the given newsgroup. If the username is not in the file **LIBDIR/authorized** then the default newsgroup pattern in the symbol **FASCIST** is used.

The format of the file *authorized* is:

user:allowed groups

For example:

root:net.all,mod.all

naughty\_person:junk,net.politics

operator:net.all,general,test,mod.unix

An open environment could have **FASCIST** set to all and then individual entries could be made in the authorized file to prevent certain individuals from posting to such a wide area.

†UNIX is a trademark of AT&T Bell Laboratories.

Note that a distribution of all does *not* mean to allow postings only to local groups – all includes all.all. Use all.all.all to get that behavior

### 2.1.39. SMALL\_ADDRESS\_SPACE

Define this if your machine has 16 bit (or smaller) pointers. If you are on a PDP-11†, this is automatically defined.

## 2.2. Makefile

There are also a few parameters in the *Makefile* as well. These are:

### 2.2.1. OSTYPE

This is the type of UNIX system you are using. It should be either v7 or USG. Any BSD system is v7. Any System 3 or System 5 system is USG. This is normally set by *localize.sh*.

### 2.2.2. NEWSUSR

This is the owner (user name) of *inews*. If you are a superuser, you should probably create a new user id (traditionally *news*) and use this id. If you are not a superuser, you can use your own user id. If you are able to, you should create a mail alias *usenet* and have mail to this alias forwarded to you. This will make it easier for other sites to find the right person in the presence of changing jobs and out of date or nonexistent directory pages. NEWSUSR and ROOTID do not need to represent the same user.

### 2.2.3. NEWSGRP

This is the group (name) to which *inews* belongs. The same considerations as NEWSUSR apply.

### 2.2.4. SPOOLDIR

This directory contains subdirectories in which news articles will be stored. It is normally */usr/spool/news*.

Briefly, for each newsgroup (say *net.general*) there will be a subdirectory */usr/spool/news/net/general* containing articles, whose file names are sequential numbers, e.g., */usr/spool/news/net/general/1*, etc.

Each article file is in a mail-compatible format. It begins with a number of header lines, followed by a blank line, followed by the body of the article. The format has deliberately been chosen to be compatible with the ARPANET standard for mail documented in RFC 822.

You should place news in an area of the disk with enough free space to hold the news you intend to keep on line. The total volume of news in *net.all* currently runs about 1 Mbyte per day. If you expire news after the default 2 weeks, you will need about 14 Mbytes of disk space (plus some extra as a safety margin and to allow for increased traffic in the future.) If you only receive some of the newsgroups, or expire news after a different interval, these figures can be adjusted accordingly.

### 2.2.5. BATCHDIR

This directory will contain the list of articles to send to each system. It is normally */usr/spool/batch*.

†PDP-11 is a trademark of Digital Equipment Corporation.





### 2.2.6. LIBDIR

This directory will contain various system files. It is normally */usr/lib/news*.

### 2.2.7. BINDIR

This is the directory in which *readnews*, *postnews*, *vnews*, and *checknews(1)* are to be installed. This is normally */usr/bin*. If you decide to set **BINDIR** to a local binary directory, you should consider that the *rnews* and *cunbatch* commands must be in a directory that can be found by *uuxqt*, which normally only searches */bin* and */usr/bin*.

### 2.2.8. UUXFLAGS

These are the flags *uux* will be called with.

### 2.2.9. LNRNEWS

This is the program used to link *rnews* and *inews*. If you have symbolic links, you can replace the "ln" with "ln -s".

### 2.2.10. SCCSID

If this is defined, *scs* ids will be included in each file. If you are short on address space, don't define this.

## 3. FILES

This section lists the files in **LIBDIR** and comments briefly what they do.

### 3.1. active

A list of active newsgroups. It is automatically updated as new newsgroups come in. The order here is the order news is initially presented by *readnews*, so you can edit this file to put important newsgroups first. If you have **SORTACTIVE** defined, after the first time the user invokes *readnews*, it will be presented in the order of his *.newsrsc*. Each line of the active file contains four fields, separated by a space: the newsgroup name, the highest local article number (for the most recently received article), the lowest local article number that has not yet expired, and a single character used to determine if the user can post to that newsgroup. If the character is "y" the user is permitted to post articles to that group. If the character is "n" the user is not permitted to post articles to that groups. (This field takes the place of the *ngfile* in earlier versions of news. Local article numbers begin at 1 and count sequentially within the newsgroup as articles are received. They do not usually correspond to local article numbers on other sites. The article numbers are always stored as a five digit number (with leading zeros) to allow updating of the file in place.

The active file should contain all active net-wide active newsgroups (*net.all* and *mod.all*). It is important that they all be present, as they are used as a check for valid newsgroup names and invalid newsgroup names are removed from any articles processed by *inews*. You should use the *sys* file to keep out unwanted newsgroups.

### 3.2. aliases

This file is used to map bad newsgroup names to the correct ones. (For example, *net.unix.wizards* is mapped into *net.unix-wizards*). Each line consists of two fields separated by a space. If the first field is found in the newsgroup list of the incoming article, it is changed to the second field. This change takes place in the article before it is passed on to other systems, not just locally.





### 3.3. batch

This program reads a list of filenames of articles and outputs the articles themselves. It is typically used by the shell script *sendbatch*.

### 3.4. c7unbatch

This is used to decompress news that has been encoded for transmission over a network that only supports 7-bit transfers (e.g. X.25.)

### 3.5. caesar

This is a program to do Caesar decoding of rotated text, on a line by line basis. The standard input is copied to the standard output, rotating each line according to a static single letter frequency table. If an integer argument is given (e.g., 13), every line is rotated by that argument, without regard to letter frequencies. This program is invoked by the *D readnews* command. It is also used by *postnews* with the "13" argument to encode selected material for posting.

### 3.6. checkgroups

*Checkgroups* is a shell file to aid in automatically checking the accuracy of your active file. It is executed by the *checkgroups* control message and mails a list of out of date newsgroups to the person defined by *NOTIFY*. It also updates the *newsgroups* file that is used by *postnews* as a helpfile for newsgroup selection.

### 3.7. compress

This program does a modified Lempel-Ziv data compression. It is used by the compressed batching scheme. It averages 50% compression on a typical batch of news.

### 3.8. distributions

This is a list of distributions that are valid for your site. Each line has two fields separated by the first space on the line. The first field is the name of the distribution (e.g., *usa*, *na*, etc.). The second field is text describing the distribution. As distributed, this file is only correct for sites in the USA. You should examine this file and add or delete the appropriate distributions.

### 3.9. encode

This program transforms an 8-bit binary file into a file suitable for sending over a link that only allows 7-bit characters. It is used by *sendbatch -c7*.

### 3.10. errlog

This file contains the "important" error messages found in the *log* file. These errors usually indicate that something was wrong with an article. This file should be watched closely. The *log* file contains much more verbose information and it is often difficult to detect errors in it.

### 3.11. expire

This program expires old articles and archives them if archiving is selected. It is typically run once a day from *cron*(8).

### 3.12. help

This contains a list of commands printed when an illegal command is typed to *readnews*.

**3.13. history**

A list of every article that has come in to your system. It is used to reject articles that come in for the second time (presumably via a different path). This file will grow but is cleaned out by the *expire(8)* command.

**3.14. history.d**

On USG systems, this directory contains 10 files (*history.[0-9]*) which are used as part of a simple hashing algorithm to speed up history searches. Since V7 systems have DBM, this is not used on V7 systems.

**3.15. history.dir, history.pag**

These two files are used on V7 systems as a hashed version of *history*, containing the message id's of all articles in history. They are only used if *-DDBM* and *-ldb* appear in *Makefile*.

**3.16. inews**

This is the program that actually sends and receives news. All other programs interface eventually with it. It is not intended to be used directly by a human, so it is no longer in */usr/bin*.

**3.17. log**

If present, a log of articles processed and error conditions is kept here. This file grows without limit unless cleaned out periodically. The *trimlib* script in *misc* can be invoked from *cron* daily or weekly to keep the log short.

**3.18. moderators**

This file contains a list of the moderators and their mailing addresses for each moderated newsgroup. Each line consists of two fields, the first is the name of the moderated group. The second is the mailing address of the group's moderator. As distributed, they are almost certainly wrong. You will need to modify the paths so they work from your site.

**3.19. newsgroups**

This file is displayed by *postnews* when a user hits ? in response to its request for newsgroups. It is also used by *vnews* when it displays the newsgroup name. It is updated automatically by the *checkgroups* control message.

**3.20. notify**

If this file is present, its contents will be taken as the name of the user to notify in case of a problem. If the file is empty, nobody will be notified. (This overrides the **NOTIFY** option in *defs.h*). Having a null file is useful if one person administers several systems and does not want multiple copies of control message notifications.

**3.21. oactive, ohistory, ohistory.dir, ohistory.pag**

These are copies of the corresponding *active*, *history*, *history.dir*, and *history.pag* files before *expire* ran. They are kept in case something happens to the originals.

**3.22. recmail**

This program can serve as a link between news and your local mailer. If you have *sendmail(8)*, don't use *recmail*. *Sendmail* is much more useful.

### 3.23. recnews

A program which allows you to send mail to get news posted. You usually need to run *sendmail* or *delivermail*(8) to be able to use this.

### 3.24. recording

A list of newsgroup classes and filenames to display recordings for. The recording feature is analogous to the recordings played in some areas when you dial directory assistance, trying to be annoying and make you think twice. Recordings on certain newsgroups are intended to remind the user of the rules for the newsgroup, or, in the case of a company worried about letting proprietary information out, reminding authors that anything they say is seen outside the company and so proprietary information should not be included.

The file contains one line per recording. The line contains two fields, separated by a space. The first field is the newsgroup class (*e.g.*, *net.all*), the second field is the name of the file containing the recorded message. If the file name does not begin with a slash, it will be searched for in *LIBDIR*. Sample recording files can be found in the *misc* directory.

### 3.25. rmgroup

This shell file should be used to remove any groups that are no longer used.

### 3.26. sendbatch

This shell file is used to send batched articles to other systems. It is typically run from *cron*. See the manual page for more details.

### 3.27. sendnews

A program to send news internally from one computer to another. It is useful if you must use mail links to transmit articles.

### 3.28. seq

This file contains the current sequence number for your system. It is used to generate unique article id's.

### 3.29. sys

This file contains a list of all your neighbors, which newsgroups they get, and how to send news to them. The format is documented below.

### 3.30. unbatch

This program is used to unbatch the incoming batched news and feed each article to *inews*. It's horrible and will go away in the future.

### 3.31. users

A list of users that have read news on your system.

### 3.32. uurec

A program to receive news sent by *sendnews*(8).

### 3.33. vnews.help

This is the helpfile used by *vnews*.



#### 4. Setting Up Links

There are two basic types of links for exchanging news: those that use mail and those that don't. The ones that use mail are more indirect, yet more versatile, while the ones that don't are simpler. The default method does not use mail, so that is discussed first.

##### 4.1. Non-mail Links

The basic theory behind a non-mail link is that the *rnews* program is invoked on the remote system with the article being transmitted as the standard input. This is possible on several networks, but the most common implementation is via the UUCP network. Using the *uux* command, the command which is forked to the shell looks like:

```
uux - -r -z remotesys!rnews < article
```

This is the default transmission method. In order to set up such a link, obviously a UUCP link with the remote system must be in effect. In addition, *rnews* must be available and executable by *uuxqt* on the remote machine. In most cases, this means that *rnews* must be in */usr/bin* so *uux* can find it. Also, the list of allowed UUCP commands (in */usr/src/usr.bin/uucp/uuxqt.c* or */usr/lib/uucp/L.cmds*, depending on the version of UUCP) should be checked to make sure that *rnews* is an allowed command.

Other networks that allow remote execution include the BERKNET, BLICN (*usend(1)*), many Ethernets, and the NSC hyperchannel (*nusend(1)*). It is important, however, that a spooling mechanism be available. Otherwise, if system *A* tries to send an article to system *B* via a remote execution command, and *B* is down, the article could be lost. Spooling arranges that the system will try again when *B* comes back up.

##### 4.2. Mail Links

When using mail to transmit articles, two intermediary programs are necessary. These are *sendnews* and *uurec(8)*. The idea is that when system *A* wants to send an article to system *B*, the *sys* file on system *A* has an entry for system *B* such as:

```
/usr/lib/news/sendnews -a rnews@B
```

which runs *sendnews* on the article. The *-a* option specifies that the mail should be formatted for the ARPANET. *Sendnews* packages the article and mails it to "rnews@B". Somehow, the *B* system is expected to make sure that all mail to user "rnews" is fed as input to the program *uurec*. This program unpackages it and invokes *rnews*.

The best way to get mail to "rnews" fed into *uurec* is to use *sendmail* or *delivermail*, if you are on a system running them. Create an alias in */usr/lib/aliases* as follows:

```
rnews: "|/usr/lib/news/uurec"
```

and *sendmail* will handle it. If you do not have a facility for forwarding mail to a program, you can gimmick your mailer to watch for it (using *popen(3S)*, this is easy) or, if you don't want to do any programming, you can have *cron* invoke *uurec* every hour with */usr/spool/mail/rnews* as standard input. This solution is messier because *uurec* must potentially deal with multiple messages, something that has never been tested.

#### 5. Format of the *sys* file

To set up a link to another site, edit the *sys* file in **LIBDIR**. This file is similar to the *L.sys* file of UUCP. Each line contains four fields, separated by colons:

- (1) The system name of a site to which you forward news. Normally all systems you have links to will be included. You should also have a line for your own system. If this field is *ME*, it will be used as if it were your local system name. If the system name is followed by a "/", the article will not be forwarded to this system if it has already passed through any of the (comma separated) list of sites immediately following the "/". For example, if the sysline was:



`yoursite/sitea,siteb,sitec:net,mod,na,usa,to.yoursite::`

the incoming article would only be forwarded to *yoursite* if it had not already been to any of *sitea*, *siteb*, or *sitec*. This is normally used to reduce the number of duplicate articles received at a site that has multiple main newsgroups.

- (2) The newsgroups to be forwarded to them. This is a pattern of the same kind as a subscription list. Generally, you will list classes of newsgroups, that is, using `all` for everything. A typical forwarding list for a new site would be

`net,mod,na,usa,to.sysname`

where *sysname* is the name of the remote system. (Of course, if you are not in the USA or North America, you would remove those distributions and replace them with the ones appropriate for you). In particular, you don't want to forward all since local newsgroups (those without dots) should not be sent. For the line describing your own system, this field describes the newsgroups your site will accept from remote sites. Thus, if another site insists on sending you a newsgroup you don't want, for example `net.jokes`, include `!net.jokes` here.

- (3) This field contains flags describing the connection. An `A` will indicate that the other site is running an `A` version of netnews. A `B` indicates a `B` version. Leaving it empty defaults to `B`. If you are reading this document, you have a `B` version. Some existing sites run `A` versions. If you aren't sure, ask your contact at the other site, with whom you should be talking to set this up anyway. The `F` flag indicates that the fourth field is the name of a file. The full path name of a file containing the article in `SPOOL` will be appended to this file. The `L` flag prevents transmission unless the article was created on this site. If a number follows the `L` (e.g., `L3`), sites less than that number of hops away will be considered local. (It is recommended that you feed an `L` link to a backbone site, to ensure that your submissions will be more likely to get to the entire network, even in the event of a local problem. Please make sure that a mail link exists too, so you can get replies.) The `N` flag can also be included here, indicating that mail should be sent using the *ihave/sendme* protocol described below. The `H` flag can be used to interpolate the history file into the command. The `S` flag says to execute the transmission command directly instead of forking a shell. The `U` field arranges that the parameter to the optional `%s` in the command field to be filled in with a permanent file name from `SPOOL` instead of a temporary customized file name. The `M` flag says to use multi-casting. Multi-casting is described in an appendix.
- (4) This field is the command to be run to send news to the remote site. The article will be on the standard input. Leaving this field blank means an ordinary `UUCP` link is being used, that is, the command defaults to

`uux - -r -z sysname!rnews`

The `-o` option tells `uux` to expect input from the standard input. The `-z` option is nonstandard – you should add it (see the *minus.z\** files in the `uucp` source directory.) It shuts off the annoying message you would otherwise get mailed to you telling you that your article was broadcast successfully. To avoid using the `-z` option, change the source or put the `uux` command in the fourth field. The `-r` option tells `uux` not to call the other system once the job is queued. This turns out to ease the load on the system, at the expense of making news be transmitted a bit slower. The news will be sent when the next call is made; usually this means the next time mail is sent to or from your system. If this turns out to be unreasonably long, put a line in *crontab* to run

`/usr/lib/uucp/uucico -r1 -ssystem`

every hour or so.

Here is a sample *sys* file for a site *myvax* with connections to *yourvax* where *myvax* also passes news on to *downstream*. We assume that *myvax* and *downstream* exchange a local newsgroup class `lng.all` as well as the network wide newsgroups. News to *downstream* is batched. We also assume that *myvax* and *yourvax* are in the USA, while *downstream* is in Canada.

```
myvax:net,mod,na,usa,lng,to::
yourvax:net,mod,na,usa,to.yourvax::
downstream:net,mod,na,lng,to.downstream:F:/usr/spool/batch/downstream
```

## 6. Posting Methods

The basic method is *postnews*. This program will prompt you for the title, newsgroups, and distribution, then place you in the editor. (The system default **EDITOR** is used unless the environment variable **EDITOR** is set, overriding the system default.) The text should be typed after the blank line. The title and newsgroups are available for editing at the top of the buffer. Other header lines can be added, such as an expiration date or a distribution. When you write out the file and exit from the editor, you will be prompted for what to do next. Your choices are: write the message to a file, send the message, list the message or edit it again.

Another method is to use mail. This can only be done on systems that allow mail to a given name to be fed into an arbitrary program as input. This is easily done with the Berkeley *delivermail* or *sendmail* program, and not with any other mailer the author is familiar with. (It may be possible to painfully set this up with MMDF, provided the newsgroup name is no more than 8 characters long.) To use mail, set up an alias such as the following:

```
net.general: "|/usr/lib/news/recnews net.general"
```

Whenever a user sends mail to *net.general*, this starts up the given shell command which calls *recnews* with one argument, the name of the newsgroup. You need to create one alias for each newsgroup, and to keep the list up to date as new newsgroups are created. *Recnews*(8) will in turn invoke *inews*.

Note that there are problems with *recnews*. There is no way to use it to post to multiple newsgroups without creating separate articles (something frowned upon because it forces people to read the same thing more than once.) Also, there is no way to make the recording feature (to remind people to not accidentally divulge proprietary information) work when *recnews* is used.

## 7. Various considerations

### 7.1. Setuid bits

The current intended state of affairs is that *inews* runs setuid to **NEWSUSR**. The *readnews* program does not need to be setuid. This makes it possible to write your own interface to read news instead of using *readnews*. (As distributed, *inews* is also setgid. I know of no good reason for this.)

### 7.2. Modes of Spool Directories

All the files should be writable by **NEWSUSR**. However, due to a glitch, you will probably have to make the **SPOOLDIR** and its subdirectories mode 777. It could be 755 except for one problem. When a new newsgroup comes in, *inews* will attempt to *mkdir*(1) a new subdirectory of **SPOOLDIR** for the newsgroup. Since both *inews* and *mkdir* are setuid, *mkdir* will use the uid of the person who ran *inews* instead of **NEWSUSR** when checking for permissions. If the directory mode isn't 777 the check will fail. Here are several alternatives if you don't want a 777 directory around:

#### 7.2.1. Fix Real Uid

If *inews* is always run by *cron* or as *root*, the real uid can be arranged to be *root* or **NEWSUSR**. This is a poor solution since it makes the local creation of new newsgroups require super user permissions, and is a potential security hole. If this approach is taken, care must be taken to insure that the owner of the created directory is **NEWSUSR**.

#### 7.2.2. Change the Kernel

*Inews* will do: *setuid(geteuid())* (see *setuid(2)* and *geteuid(2)*) before it forks the *mkdir*. If your system permits this call, there will be no problem. In particular, Berkeley 4.0 UNIX and later systems allow this. An alternative change to the kernel is to automatically stack uids: when a setuid program

is run, set the new real uid to the old effective uid.

#### 7.2.3. Groups

You could have *inews* be setgid to **NEWSGRP** and all files writable by the group. This approach has been tested and the problem turns out to be that the *mkdir* command uses the *access(2)* system call to check permissions. Since *access* uses the real gid, you run into the same problem.

#### 7.2.4. Another *Mkdir*

You could create a version of *mkdir* that does less checking and put it in a directory that can only be accessed by **NEWSUSR** (mode 700, owned by **NEWSUSR**). Have *inews* fork this *mkdir*.

#### 7.3. Expiration dates

To get articles to expire automatically, put a line in *crontab* to run

```
/usr/lib/news/expire
```

every night. This command deletes all expired news. The *-a newsgroups* option causes all expired news to be archived under */usr/spool/oldnews* depending on which newsgroups are selected. (See *expire(8)* for details.)

Sometimes news is not expired when it should be. Be sure to check that *expire* has permissions to unlink files, and that it is properly setuid to **NEWSUSR**. You can manually invoke *expire* with the *-v* (verbose) option to find out what it's doing. Adding levels of verbosity (e.g., *-v6*) will get more and more output.

#### 7.4. Version to Version

Version B will understand incoming news in either version A or B format, automatically (presuming **OLD** is defined in *defs.h*.) Version B will generate either format, depending on the flag in the third field of the *sys* line. Version A will not understand version B format. Thus, it is possible for two version B sites to communicate using version A format. This will work but is not a good idea, since the translation from B to A loses information (such as the expiration date) which will not be there when translated back to version B.

News from versions A and 2.9 B do not conform to the USENET interchange standard. 2.10 B supports the standard and will communicate with either A or 2.9 B news. A news is written (losing other header information) if A is in the flags for the system. If **OLD** is defined, 2.10 will write out headers with both standard ("Date" "Message-ID") and 2.9 ("Posted" "Article-I.D.") lines so that either B system will properly handle the article. Incoming news is recognized by the first letter (A for A news), or the lack of an "@" in the "From" line (2.9). Missing fields are constructed as well as possible from the available information.

#### 7.5. Presentation Order

The order of the newsgroups listed in *LIBDIR/active* is the order the newsgroups will be presented in initially. If **SORTACTIVE** is defined in *defs.h*, after the first time news will be presented in the order of the person's *news.rc*. Initially this will be directory order, but you can edit important newsgroups like *general* to the top.

A recommended order to maintain your active file in is this:



```

net.announce.newusers
general
local.general
net.announce
local newsgroups in alphabetical order
mod.all newsgroups in alphabetical order
net.all newsgroups in alphabetical order
test
all.test
to.all
control
junk

```

## 8. Control Messages

Some news systems will send you articles that are not for human consumption. They are messages to your news system called *control messages*. Such messages contain the "Control" header. Older systems use newsgroups matching `all.all.ctl`, and this will still work, although the "Control" header is preferred. Since the newsgroup name is used for distribution only, and is not checked to ensure it's in the active file, such newsgroup names can still be used. This makes it possible to post network wide control messages with `net.msg.ctl` (or restricted broadcast such as `bt1.msg.ctl`) or messages for a particular system: `to.ucbvax.ctl`. Messages are canceled, however, with a "Control" line in a message to the same newsgroup(s) as the original message.

A control message contains a command and zero or more arguments (much like a UNIX program). The subject of the article contains the command and arguments. The body of the article is usually ignored, although some messages can use it for additional text information. Control messages are not stored in **SPOOL**; rather, they are acted on and discarded at once.

### 8.1. ihave/sendme

Two control messages are `ihave` and `sendme`. These messages allow two participating sites to set up a link so that one site will tell the other site it has a given article and wait for a request before it actually sends it. The normal case is to send an entire article to a system, which consults the history file to see if the article has already been seen, and then throws it away if it has been seen before.

Note that, since most messages are short anyway, experience has indicated that for ordinary UUCP unbatched communication, all *ihave/sendme* does is triple the load and slow down forwarding. We hope future code will allow `ihave`'s with multiple message id's in the body, and existing code in 2.10 understands such messages, but does not generate them. So we advise that you don't use *ihave/sendme* for now.

Use of these control messages can cut down on this wasted transmission, but if you have a polled UUCP connection, they can slow down receipt of news due to polling delays. It is up to each connected pair of sites whether they want to use this protocol. The choice is controlled by the `N` flag in the `sys` file. In the case of a leaf node (one with only one neighbor) there is no advantage to this protocol. Even if both sites are able to initiate a connection (have dialers or the link is hardwired) the `-r` option on the `uux` can cause 2 hour or more delays in propagating news. Since this protocol can triple the number of messages generated, you should carefully evaluate your situation when deciding whether to use it. If transmission time and phone bills dominate your costs, and you are sending news to several sites, and large article bodies dominate the costs (rather than the headers and the time spent by UUCP negotiating transmission) it is probably worthwhile to use *ihave/sendme*. If your costs are dominated by CPU load from UUCP, or if you send news to a site that cannot get it from anywhere else, you probably do not want to use this protocol. The decision can be made independently for each site in your `sys` file.

This pair works as follows: Site *mysite* receives article "<123@abc.UUCP>". It enters it locally and then broadcasts it to its neighbors. One of its neighbors is site *yoursite* which has the `N` flag in



the *sys* file. So *mysite* sends an article on newsgroup *to.yoursite.ctl* with title "ihave <123@abc.UUCP> mysite". This control message has two arguments - the first ("<123@abc.UUCP>") is the article id of the article in question, the second ("mysite") is the name of the site sending the article. The name of the newsgroup and the *sys* file control transmission of the article. Normally the *sys* file will read something like

yoursite:net.all,fa.all,to.yoursite:BN:

which will cause an article on *to.yoursite.ctl* to be transmitted.

*Yoursite* receives the message and looks to see if it has seen it before. If it has, it throws the message away and stops. If it hasn't, it sends a message on *to.mysite.ctl* with title "sendme <123@abc.UUCP> yoursite" which is transmitted to *mysite*. (The two arguments to *sendme* are the article id requested and the site to send it to.) Then *mysite* gets this message and actually transmits the article to *yoursite*.

## 8.2. newgroup

This message has one argument, the name of a newsgroup to be created. This allows special action to be taken locally when a new newsgroup is created. It is generated by the *-C* option to *inews*. By default, the newsgroup is added to the active file, and mail is sent to the local contact advising that this has happened. The directory will be created when a message for that newsgroup arrives. See the routine "c\_newgroup" in *control.c* if you want something different to happen. (Note that, although the body of the message contains a brief description of the purpose of the group, this body is usually thrown away by existing software.)

## 8.3. rmgroup

This message has one argument, the name of a newsgroup to be removed. It is used for network-wide cancellation of a newsgroup. If *MANUALLY* is not defined, it will remove the articles, directory, and active file line for the group. There is a shell script *rmgroup* that does essentially the same thing as this message, but the shell script only removes the group locally. We recommend that you leave *MANUALLY* defined, and when you receive mail advising you of the demise of the newsgroup, you run *rmgroup* by hand. This will prevent accidental or malicious removal of a good newsgroup.

## 8.4. cancel

This message cancels a given article. It takes one argument, the message id of the article to cancel. It should be broadcast to the same newsgroup as the original article. If the article to be canceled is not present, the control message will not be propagated to downstream sites.

## 8.5. sendsys

The *sys* file is mailed to the originator of the message. There are no arguments. This is used for making maps. Since your *sys* file is public information, you should not remove or change this control message.

## 8.6. senduname

The *uname* program is run and the output is mailed to the originator of the message. There are no arguments. This is used for making UUCP maps. If you do not run UUCP or have sites in your *L.sys* which are a secret, you may wish to edit this. Note that only the output of *uname* is mailed, not the contents of *L.sys* (which news does not have access to anyway). If you do make a change, you should arrange that some mail still is sent out to the originator of the message, so he will know your site received it. See the code in routine "c\_senduname" in *control.c*.



### 8.7. version

The local version name/number of the netnews software is mailed back to the author of the control message.

### 8.8. checkgroups

This control message is an attempt at semi-automatic maintenance of the list of active news groups. This control message takes the body of the article and pipes it into *LIB/checkgroups*. As mentioned previously, *LIB/checkgroups* will update the newsgroups file, add any missing newsgroups, and mail a message to NOTIFY about any old newsgroups that should be removed. It is expected that the person who maintains the list of active newsgroups will broadcast this control message on a regular basis.

### 8.9. Other Messages

Any unrecognized message will cause an error message to be mailed to the local site administrator. Additional messages may be defined as time goes on, such as messages to automatically update directories or maps. You should be willing to go into the code (*control.c*) and add messages as they become standardized.

## 9. Maintenance

There are some things you should do periodically to keep your news system running smoothly. We hope to eventually automate all or most of this, but right now some of it must be done by hand.

The *history* and *log* files in your *LIB* directory will grow. You should make sure that they are cleaned up periodically. The *LIB/expire* program will remove lines from history corresponding to deleted articles, but it is a good idea to check the file every few months to make sure it is not going wild. Be sure not to completely lose your history file when you clean it up, in case another neighbor tries to send you an article you recently got. (If you only get news from one site it is safe to clean it out completely.)

The log file is not automatically cleaned out by any netnews software, and will grow quickly. The *misc/trimlib* script can be installed in *LIB/trimlib*, and invoked weekly by *cron*.

You should also clean out old newsgroups that are no longer active. To remove a newsgroup *net.foo*, you should run the shell script *rmgroup* with *net.foo* as the argument. That is,

```
/usr/lib/news/rmgroup net.foo
```

Note that clearing up UUCP constipation is another thing you'll have to do if you have flaky hardware or phone lines. If you have more than one connection, chances are that UUCP will get clogged up when one of your neighbors goes down for more than a few hours. Various spooling schemes are being worked on to help make the news/uucp system more robust, but one thing you can and should do, if you find your */usr/spool/uucp* directory getting too big, is to install a subdirectory fix to UUCP. A quick and dirty version of this is available from Duke, which traps the file-oriented system calls at the assembly language level and maps, for example, *D.fooA1234* into *D.foo/D.fooA1234*. Since the *C.* and *D.local* directories still get big, in practice this can still create some big directories, but the directories tend to be a factor of 5 smaller, resulting in a factor of 25 improvement to speed (since a directory traversal for all files is quadratic on UNIX). Right now, UUCP is the weak link in netnews distribution, and you should certainly keep an eye on it.

## 10. Creating New Newsgroups

As system news administrator, you are able to create newsgroups. To create a newsgroup, first make sure this is the right thing to do. Normally a suggestion is first posted to *net.news.group*, *net.relatedgroup* for a net newsgroup *net.relatedgroup* (should be the group which you are proposing to sub-divide. For instance, to propose creating *net.tv.soaps*, post the original article to *net.tv.net.news.group*). Followups are made to *net.news.group only*. (You can force this by putting the

line:

Followup-To: net.news.group

in the headers of your original posting). If it is established that there is general interest in such a group, and a name is agreed on, then someone creates it by typing the command

`inews -C newsgroup`

This will create the active entry locally. The directory will be created automatically when the first article for that newsgroup is received. It will also prompt you for a paragraph describing the group and start up an *inews* to post a newsgroup control message announcing the group. This control message will be sent out on `net.msg.ctl` and other sites may have configured their systems to do something with these messages. A human readable announcement is not made – you can post this to `net.news.group` if necessary.

You must be the super user to use the `-C` option to *inews*. (That is, your uid must match **ROOTID**. It is recommended that you change **ROOTID** to your own uid so you don't have to *su* to create newsgroups.)

## 11. Conversion from A to B

If you are currently running version A on your system, note that B is incompatible with A. The files are stored in a different format (headers have mail like field names now). The directory organization is different (each newsgroup has a subdirectory of its own, and the file names are numbers rather than *site.id* pairs). There are no *bitmap*, *uindex*, or *nindex* files to be trashed (which articles have been read is stored in each users *.newsrsc* file). The user interface is slightly different (*news/netnews(1)* is now called *readnews*, news is posted using *inews*, subscription is done by editing *.newsrsc*, the sense of the `-c` option is reversed, news is presented in newsgroup order, the `-a` and `-t` options now probably need `-x` as well, and there are many minor changes).

We decided not to provide a program to convert from version A to version B. Rather, the following strategy was adopted for conversion:

- (1) Install the new news in a different spool directory from the old one. For example, you can use */usr/spool/newnews*. You can change to the standard name later if you want. Get it to work for local messages.
- (2) Post an article to newsgroup *general* with the old news announcing the change. Make available documentation such as the accompanying paper *How to Read the Network News* to the users. This article will be the last one in the old news.
- (3) *Chmod* the old news directory to 555 to prevent any more news from being posted. (Actually, this will prevent the bitfile from being updated, so it may not be a good idea.)
- (4) Replace the old *rnews* program with the new *rnews* program.
- (5) Test it by having your neighbor send you a message.
- (6) Wait a reasonable period for everyone to have read the final article with the old news. Perhaps a few weeks is right.
- (7) Uninstall the old news.

Users will have to invoke *readnews* instead of *netnews* to read news. Depending on your old method of posting, this could be changed too. (If you were using mail, it does not need to be changed.) They will also have to fix their subscriptions. In general, they can type

`netnews -s`

to see what they subscribe to on the old system, and then create a file in their home directory called *.newsrsc* containing

`options -n their subscription`

The format of the subscription pattern matching is the same as in A except that **ALL** is replaced by **all** (change to lower case). Something along the lines of this could be used to automate this:



```
(echo -n "options -s" ; netnews -s | sed s/ALL/all/) > .newsrsc
```

## 12. Conversion from 2.9 to 2.10

Conversion from 2.9 to 2.10 is not nearly as involved as an A to B conversion. The user interface does not change much, and the user *.newsrsc* files are not affected. However, it is recommended that you do the conversion during a time when no news is received, so that incoming news will not get lost. One way to ensure this is to make */usr/bin/rnews* be a shell script which saves the article in */usr/spool/innews/\$\$* (*\$\$* is the process id of the particular shell and will be unique for each article).

The first step to conversion is to customize the sources. In the past, you had to take a fresh distribution and edit the *defs.h* file and *Makefile* to suit local preferences. If you had many local changes, or didn't record the local changes, upgrading could be annoying. 2.10 provides a mechanism to automate these changes. Create a shell script in the *src* directory called *localize.sh*. (You can use *localize.sample* as a template.) This shell script should copy *defs.dist* to *defs.h*, and copy either *Makefile.v7* or *Makefile.usg* to *Makefile*. It should *chmod* any files that need to be changed (often *Makefile* and *defs.h*) to a writable mode. Then it should invoke *ed(1)* on the files, making any necessary local changes.

The next step is to compile the software, with *make(1)*. It may be necessary to update the *localize.sh* file until you are satisfied with the compilation. Note that after any change to the *Makefile* in *localize.sh*, you should run *localize.sh* by hand. Otherwise, although *make* will run it for you, it will then continue to do the *make* with the old *Makefile*.

When the software is compiled, you should run the *cvt.active.sh* shell script, with the *lib* and *spool* directories as parameters. This will create a new active file in *LIB/active*. Then run *cvt.links.sh* with the *lib* and *spool* directories as parameters. Then run *cvt.names.sh* with the *lib* and *spool* directories as parameters. Old news will be linked into the new hierarchy while leaving links in the old hierarchy. If you were using the default library and spool directories, you would do the following:

```
sh cvt.active.sh /usr/lib/news /usr/spool/news
sh cvt.links.sh /usr/lib/news /usr/spool/news
sh cvt.names.sh /usr/lib/news /usr/spool/news
```

The next step is to back up the old binaries:

```
mv /usr/bin/rnews /usr/bin/ornews
```

...

and to install 2.10 with

```
make install
```

Once it is installed, any incoming news will be placed into the new hierarchy but not the old one. The critical time window is between running the three shell files and installing the new software – any incoming news between these two points will appear in only the old hierarchy and be lost to the new software. If any significant time elapses here, you should divert *news* into a separate spool directory as described above.

It is crucial that you run *expire* before any new news arrives. *Expire* will update several key files automatically.

Finally, test things by posting articles to *to.neighbor* newsgroups and watching some incoming news, and announce the change to your users.

When you are satisfied that the conversion was successful, run the shell file *cvt.clean.sh* which will remove the old 2.9 news hierarchy.

**Appendix A: Setting up a Compressed, Batched Newsfeed**

First, **BATCH** must have been *#define'd* when you built the news system. To check, look in the file *defs.h* in the news source directory. **BATCH** should be defined as a program name (by default, *unbatch*). If it's undefined or commented out, define it, re-make the news system, and install the new software.

You'll also need a working *compress* program. Use the one shipped with this news distribution, which is based on version 4.0. Your news neighbors should be running a compatible version of *compress*. Versions 3.0 and 4.0 are compatible with each other, but both are incompatible with versions 2.0 and before.

Update your *sys* file. First, add the **F** flag to the other news system's line. For instance, if your compressed-and-batched news feed is named *frobozz*, and its *sys* file entry looks like: *frobozz:net,mod,na,usa,ca,to.frobozz:* then add the **F** flag as the third (colon-separated) field: *frobozz:net,mod,na,usa,ca,to.frobozz:F*. Now the pathnames of articles to be sent will be stashed in a file. This file is named in the fourth field of the *sys* entry; add it now. Use an entry of the form *BATCHDIR/system*, where *BATCHDIR* is usually */usr/spool/batch* (the actual value is defined in the news *Makefile*), and *system* is the name of the remote system, in this example *frobozz*. A name of that form is necessary: the *sendbatch* script, which sends the batched news, looks for a file name of this form to decide if there's news for the remote system.

Your completed *sys* file line should look something like:

```
frobozz:net,mod,na,usa,ca,to.frobozz:F:/usr/spool/batch/frobozz
```

In */usr/lib/crontab*, find or create at least two news lines: one that runs nightly, and one that runs every hour or so. The nightly-run script should run *expire*, trim log files, and perhaps compile weekly statistics that you post to a local-area newsgroup one day a week. The hourly-run script should complete the transmitting task with a line like:

```
sendbatch -c frobozz
```

Make sure the script knows how to get to the directory in which *sendbatch* lives. You can either mention the directory in the script's *PATH*-setting line, or replace *sendbatch* with its full pathname. *Sendbatch* reads the files mentioned in */usr/spool/batch/frobozz*, batches them, optionally compresses them, sends them to the remote system, and arranges for remote processing.

This remote processing is directed by another file in *BATCHDIR*. Make a file with a name of the form *BATCHDIR/system.cmd* (for this example, */usr/spool/batch/frobozz.cmd*). Put a line in it specifying the command that the remote system should execute to unpack the news batches that your system will send. An example *frobozz.cmd* would be:

```
uux -r -z -n -gd frobozz!news
```

Now your system will transmit compressed batches. The receiving side of the business is handled largely by a program called *rnews*, which will call other programs in *LIBDIR* to do additional processing on the incoming batches.

Make sure there is an executable file called *rnews* in the *BINDIR* directory (check the *Makefile* for its actual location). It must be reachable by UUCP or by whatever transport you'll use to transfer the netnews. If you defined *BINDIR* as */usr/bin*, you should have no problems because *uuxqt* can already get there. If you defined it as a different directory, you may have to teach *uuxqt* to look in that directory; accomplishing this varies from system to system. On 4.2BSD, add the directory to the *PATH=* line of your UUCP *L.cmds* file. On System V, on the *rnews* line of your *L.cmds* file, add a comma followed by the remote system's name on that line. If yours is in */usr/bin/news/rnews*, your *L.cmds* file will look like:

```
[For 4.2BSD]
PATH=/bin:/usr/bin:/usr/bin/news
rnews
```

[For System V]

`/usr/bin/news/rnews,frobozz`

Other systems have a similar file in the `/usr/lib/uucp` directory by which you can specify added programs and paths different from the defaults. HP-UX, for example, has a `/usr/lib/uucp/COMMANDS` file which expands *uuxqt*'s horizons. In more restrictive cases, paths are compiled into *uuxqt*. If you can't modify any UUCP files, just put *rnews* in `/usr/bin`.

You must also have a *cunbatch* in **LIBDIR** (wherever your *Makefile* defines it), because *rnews* will eventually try to exec that copy.

Tell the person at the other end of your newsfeed to use *sendbatch -c* to send you news. Once that's in place, watch your UUCP *LOGFILE* and your news *log* and *errlog* files to ensure that news is being correctly received and unpacked on your system.

Older compressed batching systems will try to exec *cunbatch* instead of *rnews*. If you are still communicating with these, leave *cunbatch* in **BINDIR** until they have upgraded their software.

## Appendix B: MULTICAST

If this is defined (in *defs.h*) then two new flag characters become defined in the *sys* file. The first, and most important, of these is the M flag.

If the M flag is set on some line in the *sys* file, then the fourth field (transfer command) is redefined to become a *multicast* name. That is simply another system name, expected to be found in the first field of some line in the *sys* file (textually following the line containing the M flag).

When a news item is being retransmitted, if it should (according to the subscription list) be sent to a system that has the M flag set, then instead of a command being run immediately to transmit the news, the news system remembers the system name, along with the multicast name (fourth field).

Eventually the multicast system name is found in first field of a *sys* file line. If its subscription list allows transmission of this news item, then its command will be executed. This command may have up to two “%s” substitutions in it. The second of those is replaced by the name of a file containing the news item (used with the U flag). The first is subjected to rather special treatment. The whole “word” (delimited by white space) containing that “%s” is duplicated as many times as there were systems with the M flag set that referenced this multicast name (which might be 0 times, causing that “word” to be omitted). In each of these duplicates, the “%s” is replaced by the name of a system. Note the multicast system name itself is not included in this process. Then the command is executed as usual.

The second flag available if the news system is built with MULTICAST defined is O. If this flag is set, then the *sys* file line will be ignored unless the system name is a multicast name from some earlier line with the M flag, and the news item is to be sent to that (earlier) system. This allows the subscription list for the multicast system name (which is likely to be a fake system name, invented just for this purpose) to be given a very wide subscription list (like all) without any unusual effects.

Here is an example. Assume that you wish to forward net.unix to four people by mail. You could do this as ...

```
fred:net.unix::mail fred
harry:net.unix::mail harry
jane:net.unix::mail jane
tony:net.unix::mail tony
```

however this causes the mail program to be started 4 times, once for each recipient. On some systems starting the mail program is a very expensive operation. If MULTICAST is defined, an alternative method is

```
fred:net.unix:M:tony
harry:net.unix:M:tony
jane:net.unix:M:tony
tony:net.unix::mail tony %s
```

This would cause just one command to be run: “mail tony fred harry jane”. Note that “tony” must still be explicitly included in the argument list to the mail command; the “%s” does not expand to include the multicast “system name” itself.

A more useful way of doing this, which does not assume that all the mail readers will want to read the same newsgroups is as follows.

```
fred:net.unix:M:Mail
harry:net.physics,net.astro:M:Mail
jane:net.unix-wizards,net.women:M:Mail
tony:net.unix,net.unix-wizards,net.jokes:M:Mail
Mail:all:O:mail %s
```

Now, if a news item in group net.unix was received, the command

```
mail fred tony
```

would be executed. If the news were in both net.unix and net.unix-wizards then the command would



be

mail fred jane tony

If a newssitem in net.med (which no-one gets by mail) arrives, then the "Mail" line will be ignored, because of the O flag. "Mail" is a fake system invented just so its "transfer command" can be used to send news to the other recipients.

The same kind of technique can be used for normal transfer of news to other systems if your transport network supports a facility to send to many other systems in one command. (That is, if it has a multicast facility.) Sun111 (the network used in Australia) has this ability, so a typical Australian sys file looks like

```
emuvax:aus.net,mod,fa:M:FakeName
kremlin:aus.net,mod:M:FakeName
kanga:aus.net,!net.all,net.unix:M:FakeName
FakeName:all:OUS:/bin/sendfile -NRSareporter -d%s -x%s
```

A news item in aus.general causes the following command

```
/bin/sendfile -NRSareporter -demuvax -dkremlin -dkanga -x/usr/spool/...
```

to be executed. Just one command is run to send the news to three remote systems.

If a multicast system has the F flag set, then the name of a file containing the news is appended to the file whose name is in the fourth field, as usual. But on the same line, separated by spaces, will be appended the names of all the systems that referenced this multicast system.

For example, if the Australian site wanted to batch news, instead of sending it directly, it would simply change the last line of its sys file to

```
FakeName:all:F:/usr/spool/batched/allsites
```

Then a news item in net.jobs would cause the following line to be appended to /usr/spool/batched/allsites

```
/usr/spool/news/net/jobs/5542 emuvax kremlin
```

This can then be processed later, in something like the normal manner. (Unfortunately no commands to do this processing are yet available).

Caution: when MULTICAST is defined, the first "%s" in all transfer commands is used for multicast, regardless of whether or not the system name is ever used as the last field of some line with the M flag set. To use the U flag in such a case, a dummy "%s" should be used, it will simply be omitted from the command that is executed.

As an example, if a sys file line were

```
foovax:net,na,usa:U:uux - foovax!foonews <%s
```

without MULTICAST, it would need to be changed to

```
foovax:net,na,usa:U:uux - foovax!foonews %s <%s
```

if MULTICAST were defined.

Additional caution: The numbers of system names that may be used in this way are quite severely restricted. Typically there may only be about 10 multicast system names, and each of those is restricted to sending to no more than about 20 systems. These limits are dynamic (that is, the numbers counted are the number of multicast systems receiving any single news item, and the number of systems that each of those will actually cause this particular news item to be sent to). These limits should easily suffice for real news sending to remote systems; however they are not likely to suffice if you want to mail news to everyone on your host.



# Name Server Operations Guide for BIND Release 4.3

Kevin J. Dunlap\*

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Sciences  
University of California  
Berkeley CA 94720



## 1. Introduction

The Berkeley Internet Name Domain (BIND) Server implements the DARPA Internet name server for the UNIX† operating system. A name server is a network service that enables clients to name resources or objects and share this information with other objects in the network. This in effect is a distributed data base system for objects in a computer network. BIND is fully integrated into 4.3BSD network programs for use in storing and retrieving host names and address. The system administrator can configure the system to use BIND as a replacement to the original host table lookup of information in the network hosts file */etc/hosts*. The default configuration for 4.3BSD uses BIND.

## 2. Building A System with a Name Server

BIND is comprised of two parts. One is the user interface called the *resolver* which consists of a group of routines that reside in the C library */lib/libc.a*. Second is the actual server called *named*. This is a daemon that runs in the background and services queries on a given network port. The standard port for UDP and TCP is specified in */etc/services*.

### 2.1. Resolver Routines In libc

When building your 4.3BSD system you may either build the C library to use the name server resolver routines or use the host table lookup routines to do host name and address resolution. The default resolver for 4.3BSD uses the name server.

Building the C library to use the name server changes the way *gethostbyname(3N)*, *gethostbyaddr(3N)*, and *sethostent(3N)* do their functions. The name server renders *gethostent(3N)* obsolete, since it has no concept of a next line in the database. These library calls are built with the resolver routines needed to query the name server.

---

\* The author is an employee of Digital Equipment Corporation's Ultrix Engineering Advanced Development Group and is on loan to CSRG. Ultrix is a trademark of Digital Equipment Corporation.

†UNIX is a Trademark of AT&T Bell Laboratories

The *resolver* is comprised of a few routines that build query packets and exchange them with the name server.

Before building the C library, set the variable *HOSTLOOKUP* equal to *named* in */usr/src/lib/libc/Makefile*. You then make and install the C library and compiler and then compile the rest of the 4.3BSD system. For more information see section 6.6 of "Installing and Operating 4.3BSD on the VAX†".

## 2.2. The Name Service

The basic function of the name server is to provide information about network objects by answering queries. The specifications for this name server are defined in RFC882, RFC883, RFC973 and RFC974. These documents can be found in */usr/src/etc/named/doc* in 4.3BSD or *fipped* from sri-nic.arpa. It is also recommended that you read the related manual pages, *named(8)*, *resolver(3)*, and *resolver(5)*.

The advantage of using a name server over the host table lookup for host name resolution is to avoid the need for a single centralized clearinghouse for all names. The authority for this information can be delegated to the different organizations on the network responsible for it.

The host table lookup routines require that the master file for the entire network be maintained at a central location by a few people. This works fine for small networks where there are only a few machines and the different organizations responsible for them cooperate. But this does not work well for large networks where machines cross organizational boundaries.

With the name server, the network can be broken into a hierarchy of domains. The name space is organized as a tree according to organizational or administrative boundaries. Each node, called a *domain*, is given a label, and the name of the domain is the concatenation of all the labels of the domains from the root to the current domain, listed from right to left separated by dots. A label need only be unique within its domain. The whole space is partitioned into several areas called *zones*, each starting at a domain and extending down to the leaf domains or to domains where other zones start. Zones usually represent administrative boundaries. An example of a host address for a host at the University of California, Berkeley would look as follows:

*monet.Berkeley.EDU*

The top level domain for educational organizations is EDU; Berkeley is a subdomain of EDU and monet is the name of the host.

## 3. Types of Servers

There are three types of servers, Master, Caching and Remote.

### 3.1. Master Servers

A Master Server for a domain is the authority for that domain. This server maintains all the data corresponding to its domain. Each domain should have at least two master servers, a primary master and some secondary masters to provide backup service if the primary is unavailable or overloaded. A server may be a master for multiple domains, being primary for some domains and secondary for others.

---

†VAX is a Trademark of Digital Equipment Corporation

### 3.1.1. Primary

A Primary Master Server is a server that loads its data from a file on disk. This server may also delegate authority to other servers in its domain.

### 3.1.2. Secondary

A Secondary Master Server is a server that is delegated authority and receives its data for a domain from a primary master server. At boot time, the secondary server requests all the data for the given zone from the primary master server. This server then periodically checks with the primary server to see if it needs to update its data.

### 3.2. Caching Only Server

All servers are caching servers. This means that the server caches the information that it receives for use until the data expires. A *Caching Only Server* is a server that is not authoritative for any domain. This server services queries and asks other servers, who have the authority, for the information needed. All servers keep data in their cache until the data expires, based on a time to live field attached to the data when it is received from another server.

### 3.3. Remote Server

A Remote Server is an option given to people who would like to use a name server on their workstation or on a machine that has a limited amount of memory and CPU cycles. With this option you can run all of the networking programs that use the name server without the name server running on the local machine. All of the queries are serviced by a name server that is running on another machine on the network.

## 4. Setting up Your Own Domain

When setting up a domain that is going to be on a public network the site administrator should contact the organization in charge of the network and request the appropriate domain registration form. An organization that belongs to multiple networks (such as CSNET, DARPA Internet and BITNET) should register with only one network.

The contacts are as follows:

### 4.1. DARPA Internet

Sites that are already on the DARPA Internet and need information on setting up a domain should contact `HOSTMASTER@SRI-NIC.ARPA`. You may also want to be placed on the BIND mailing list, which is a mail group for people on the DARPA Internet running BIND. The group discusses future design decisions, operational problems, and other related topic. The address to request being placed on this mailing list is:

`bind-request@ucbarpa.Berkeley.EDU`.

### 4.2. CSNET

A CSNET member organization that has not registered its domain name should contact the CSNET Coordination and Information Center (CIC) for an application and information about setting up a domain.



An organization that already has a registered domain name should keep the *CIC* informed about how it would like its mail routed. In general, the *CSNET* relay will prefer to send mail via *CSNET* (as opposed to *BITNET* or the *Internet*) if possible. For an organization on multiple networks, this may not always be the preferred behavior. The *CIC* can be reached via electronic mail at *cic@sh.cs.net*, or by phone at (617) 497-2777.

#### 4.3. BITNET

If you are on the BITNET and need to set up a domain, contact *INFO@BITNIC*.

### 5. Files

The name server uses several files to load its data base. This section covers the files and their formats needed for *named*.

#### 5.1. Boot File

This is the file that is first read when *named* starts up. This tells the server what type of server it is, which zones it has authority over and where to get its initial data. The default location for this file is */etc/named.boot*. However this can be changed by setting the *BOOTFILE* variable when you compile *named* or by specifying the location on the command line when *named* is started up.

##### 5.1.1. Domain

The line in the boot file that designates the default domain for the server looks as follows:

```
domain          Berkeley.Edu
```

The name server uses this information when it receives a query for a name without a ".". When it receives one of these queries, it appends the name in the second field to the query name.

##### 5.1.2. Primary Master

The line in the boot file that designates the server as a primary server for a zone looks as follows:

```
primary         Berkeley.Edu /etc/ucbhosts
```

The first field specifies that the server is a primary one for the zone stated in the second field. The third field is the name of the file from which the data is read.

##### 5.1.3. Secondary Master

The line for a secondary server is similar to the primary except for the word *secondary* and the third field.

```
secondary      Berkeley.Edu 128.32.0.10 128.32.0.4
```

The first field specifies that the server is a secondary master server for the zone stated in the second field. The rest of the line, lists the network addresses for the name servers that are primary for the zone. The secondary server gets its data across the network from the listed servers. Each server is tried in the order listed until it successfully receives the data from a listed server.

#### 5.1.4. Caching Only Server

You do not need a special line to designate that a server is a caching server. What denotes a caching only server is the absence of authority lines, such as *secondary* or *primary* in the boot file.

All servers should have a line as follows in the boot file to prime the name servers cache:

```
cache /etc/named.ca
```

For information on cache file see section on *Cache Initialization*.

#### 5.1.5. Remote Server

To set up a host that will use a remote server instead of a local server to answer queries, the file */etc/resolv.conf* needs to be created. This file designates the name servers on the network that should be sent queries. It is not advisable to create this file if you have a local server running. If this file exists it is read almost every time *gethostbyname()* or *gethostbyaddr()* is called.

### 5.2. Cache Initialization

#### 5.2.1. named.ca

The name server needs to know the server that is the authoritative name server for the network. To do this we have to prime the name server's cache with the address of these higher authorities. The location of this file is specified in the boot file. This file uses the Standard Resource Record Format covered further on in this paper.

### 5.3. Domain Data Files

There are three standard files for specifying the data for a domain. These are *named.local*, *hosts* and *hosts.rev*. These files use the Standard Resource Record Format covered later in this paper.

#### 5.3.1. named.local

This file specifies the address for the local loopback interface, better known as *localhost* with the network address 127.0.0.1. The location of this file is specified in the boot file.

#### 5.3.2. hosts

This file contains all the data about the machines in this zone. The location of this file is specified in the boot file.

#### 5.3.3. hosts.rev

This file specifies the IN-ADDR.ARPA domain. This is a special domain for allowing address to name mapping. As internet host addresses do not fall within domain boundaries, this special domain was formed to allow inverse mapping. The IN-ADDR.ARPA domain has four labels preceding it. These labels correspond to the 4 octets of an Internet address. All four octets must be specified even if an octet is zero. The Internet address 128.32.0.4 is located in the domain 4.0.32.128.IN-ADDR.ARPA. This reversal of the address is awkward to read but allows for the natural grouping of hosts in a network.



#### 5.4. Standard Resource Record Format

The records in the name server data files are called resource records. The Standard Resource Record Format (RR) is specified in RFC882 and RFC973. The following is a general description of these records:

*(name) {ttl} addr-class Record Type Record Specific data*

Resource records have a standard format shown above. The first field is always the name of the domain record. For some RR's the name may be left blank; in that case it takes on the name of the previous RR. The second field is an optional time to live field. This specifies how long this data will be stored in the data base. By leaving this field blank the default time to live is specified in the *Start Of Authority* resource record (see below). The third field is the address class; there are currently two classes: *IN* for internet addresses and *ANY* for all address classes. The fourth field states the type of the resource record. The fields after that are dependent on the type of the RR. Case is preserved in names and data fields when loaded into the name server. All comparisons and lookups in the name server data base are case insensitive.

The following characters have special meanings:

- . A free standing dot in the name field refers to the current domain.
- @ A free standing @ in the name field denotes the current origin.
- .. Two free standing dots represent the null domain name of the root when used in the name field.
- \X Where X is any character other than a digit (0-9), quotes that character so that its special meaning does not apply. For example, "\" can be used to place a dot character in a label.
- \DDD Where each D is a digit, is the octet corresponding to the decimal number described by DDD. The resulting octet is assumed to be text and is not checked for special meaning.
- () Parentheses are used to group data that crosses a line. In effect, line terminations are not recognized within parentheses.
- ; Semicolon starts a comment; the remainder of the line is ignored.
- \* An asterisk signifies wildcarding.

Most resource records will have the current origin appended to names if they are not terminated by a ".". This is useful for appending the current domain name to the data, such as machine names, but may cause problems where you do not want this to happen. A good rule of thumb is that, if the name is not in of the domain for which you are creating the data file, end the name with a ".".

##### 5.4.1. \$INCLUDE

An include line begins with \$INCLUDE, starting in column 1, and is followed by a file name. This feature is particularly useful for separating different types of data into multiple files. An example would be:

```
$INCLUDE /usr/named/data/mailboxs
```

The line would be interpreted as a request to load the file */usr/named/data/mailboxes*. The \$INCLUDE command does not cause data to be loaded into a different zone or tree. This is simply a way to allow data for a given zone to be organized in separate files. For example, mailbox data might be kept

separately from host data using this mechanism.

#### 5.4.2. \$ORIGIN

The origin is a way of changing the origin in a data file. The line starts in column 1, and is followed by a domain origin. This is useful for putting more than one domain in a data file.

#### 5.4.3. SOA - Start Of Authority

<i>name</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>SOA</i>	<i>Origin</i>	<i>Person in charge</i>
@		IN	SOA	ucbvax.Berkeley.Edu.	kjd.ucbvax.Berkeley.E
			1.1	; Serial	
			3600	; Refresh	
			300	; Retry	
			3600000	; Expire	
			3600 )	; Minimum	

The *Start of Authority*, *SOA*, record designates the start of a zone. The name is the name of the zone. Origin is the name of the host on which this data file resides. Person in charge is the mailing address for the person responsible for the name server. The serial number is the version number of this data file, this number should be incremented whenever a change is made to the data. The name server cannot handle numbers over 9999 after the decimal point. The refresh indicates how often, in seconds, a secondary name servers is to check with the primary name server to see if an update is needed. The retry indicates how long, in seconds, a secondary server is to retry after a failure to check for a refresh. Expire is the upper limit, in seconds, that a secondary name server is to use the data before it expires for lack of getting a refresh. Minimum is the default number of seconds to be used for the time to live field on resource records. There should only be one *SOA* record per zone.

#### 5.4.4. NS - Name Server

<i>{name}</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>NS</i>	<i>Name servers name</i>
		IN	NS	ucbarpa.Berkeley.Edu.

The *Name Server* record, *NS*, lists a name server responsible for a given domain. The first name field lists the domain that is serviced by the listed name server. There should be one *NS* record for each Primary Master server for the domain.

#### 5.4.5. A - Address

<i>{name}</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>A</i>	<i>address</i>
ucbarpa		IN	A	128.32.0.4
		IN	A	10.0.0.78

The *Address* record, *A*, lists the address for a given machine. The name field is the machine name and the address is the network address. There should be one *A* record for each address of the machine.

#### 5.4.6. HINFO - Host Information

<i>{name}</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>HINFO</i>	<i>Hardware</i>	<i>OS</i>
		ANY	HINFO	VAX-11/780	UNIX

*Host Information* resource record, *HINFO*, is for host specific data. This lists the hardware and operating system that are running at the listed host. It should be noted that only a single space separates the hardware info and the operating system info. If you want to include a space in the machine name you must quote the name. Host information is not specific to any address class, so *ANY* may be used for the address class. There should be one *HINFO* record for each host.

#### 5.4.7. WKS - Well Known Services

(name)	(ttl)	addr-class	WKS	address	protocol	list of services
		IN	WKS	128.32.0.10	UDP	who route timed domain
		IN	WKS	128.32.0.10	TCP	( echo telnet discard sunrpc sftp uucp-path systat daytime netstat qotd nntp link chargen ftp auth time whois mtp pop rje finger smtp supdup hostnames domain nameserver )

The *Well Known Services* record, *WKS*, describes the well known services supported by a particular protocol at a specified address. The list of services and port numbers come from the list of services specified in */etc/services*. There should be only one *WKS* record per protocol per address.

#### 5.4.8. CNAME - Canonical Name

aliases	(ttl)	addr-class	CNAME	Canonical name
ucbmonet		IN	CNAME	monet

*Canonical Name* resource record, *CNAME*, specifies an alias for a canonical name. An alias should be unique and all other resource records should be associated with the canonical name and not with the alias. Do not create an alias and then use it in other resource records.

#### 5.4.9. PTR - Domain Name Pointer

name	(ttl)	addr-class	PTR	real name
7.0		IN	PTR	monet.Berkeley.Edu.

A *Domain Name Pointer* record, *PTR*, allows special names to point to some other location in the domain. The above example of a *PTR* record is used in setting up reverse pointers for the special *IN-ADDR.ARPA* domain. This line is from the example *hosts.rev* file. *PTR* names should be unique to the zone.

#### 5.4.10. MB - Mailbox

name	(ttl)	addr-class	MB	Machine
miriam		IN	MB	vineyd.DEC.COM.

*MB* is the *Mailbox* record. This lists the machine where a user wants to receive mail. The name field is the users login; the machine field denotes the machine to which mail is to be delivered. Mail Box names should be unique to the zone.



## 5.4.11. MR - Mail Rename Name

<i>name</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>MR</i>	<i>corresponding MB</i>
Postmistress		IN	MR	miriam

*Main Rename, MR*, can be used to list aliases for a user. The name field lists the alias for the name listed in the fourth field, which should have a corresponding *MB* record.

## 5.4.12. MINFO - Mailbox Information

<i>name</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>MINFO</i>	<i>requests</i>	<i>maintainer</i>
BIND		IN	MINFO	BIND-REQUEST	kjd.Berkeley.Edu.

*Mail Information* record, *MINFO*, creates a mail group for a mailing list. This resource record is usually associated with a mail group *Mail Group*, but may be used with a *Mail Box* record. The *name* specifies the name of the mailbox. The *requests* field is where mail such as requests to be added to a mail group should be sent. The *maintainer* is a mailbox that should receive error messages. This is particularly appropriate for mailing lists when errors in members names should be reported to a person other than the sender.

## 5.4.13. MG - Mail Group Member

<i>{mail group name}</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>MG</i>	<i>member name</i>
		IN	MG	Bloom

*Mail Group, MG* lists members of a mail group.

An example for setting up a mailing list is as follows:

Bind	IN	MINFO	Bind-Request	kjd.Berkeley.Edu.
	IN	MG	Ralph.Berkeley.Edu.	
	IN	MG	Zhou.Berkeley.Edu.	
	IN	MG	Painter.Berkeley.Edu.	
	IN	MG	Riggle.Berkeley.Edu.	
	IN	MG	Terry.pa.Xerox.Com.	

## 5.4.14. MX - Mail Exchanger

<i>name</i>	<i>{ttl}</i>	<i>addr-class</i>	<i>MX</i>	<i>preference value</i>	<i>mailer exchanger</i>
Munnari.OZ.AU.		IN	MX	0	Seismo.CSS.GOV.
*.IL.		IN	MX	0	RELAY.CS.NET.

*Main Exchanger* records, *MX*, are used to specify a machine that knows how to deliver mail to a machine that is not directly connected to the network. In the first example, above, Seismo.CSS.GOV. is a mail gateway that knows how to deliver mail to Munnari.OZ.AU. but other machines on the network can not deliver mail directly to Munnari. These two machines may have a private connection or use a different transport medium. The preference value is the order that a mailer should follow when there is more than one way to deliver mail to a single machine. See RFC974 for more detailed information.

Wildcard names containing the character "\*" may be used for mail routing with *MX* records. There are likely to be servers on the network that simply state that any mail to a domain is to be routed through a relay. Second example, above, all mail to hosts in the domain IL is routed through RELAY.CS.NET. This is done by creating a wildcard resource record, which states that \*.IL has an *MX* of



RELAY.CS.NET.

## 5.5. Sample Files

The following section contains sample files for the name server. This covers example boot files for the different types of servers and example domain data base files.

### 5.5.1. Boot File

#### 5.5.1.1. Primary Master Server

```

;
; Boot file for Primary Master Name Server
;
; type      domain          source file or host
;
domain      Berkeley.Edu
primary     Berkeley.Edu    /etc/ucbhosts
cache       .               /etc/named.ca
primary     32.128.in-addr.arpa /etc/ucbhosts.rev
primary     0.0.127.in-addr.arpa /etc/named.local

```

#### 5.5.1.2. Secondary Master Server

```

;
; Boot file for Primary Master Name Server
;
; type      domain          source file or host
;
domain      Berkeley.Edu
secondary   Berkeley.Edu    128.32.0.4 128.32.0.10 128.32.136.22
cache       .               /etc/named.ca
secondary   32.128.in-addr.arpa 128.32.0.4 128.32.0.10 128.32.136.22
primary     0.0.127.in-addr.arpa /etc/named.local

```

#### 5.5.1.3. Caching Only Server

```

;
; Boot file for Primary Master Name Server
;
; type      domain          source file or host
;
domain      Berkeley.Edu
cache       .               /etc/named.ca
primary     0.0.127.in-addr.arpa /etc/named.local

```



**5.5.2. Remote Server****5.5.2.1. /etc/resolv.conf**

```
domain Berkeley.Edu
nameserver 128.32.0.4
nameserver 128.32.0.10
```

**5.5.3. named.ca**

```
;
; Initial cache data for root domain servers.
;
.          99999999 IN NS USC-ISIC.ARPA.
          99999999 IN NS USC-ISIB.ARPA.
          99999999 IN NS BRL-AOS.ARPA.
          99999999 IN NS SRI-NIC.ARPA.
; Prep the cache (hotwire the addresses).
SRI-NIC.ARPA. 99999999 IN A 10.0.0.51
USC-ISIB.ARPA. 99999999 IN A 10.3.0.52
USC-ISIC.ARPA. 99999999 IN A 10.0.0.52
BRL-AOS.ARPA. 99999999 IN A 128.20.1.2
BRL-AOS.ARPA. 99999999 IN A 192.5.22.82
```

**5.5.4. named.local**

```
@ IN SOA ucbvax.Berkeley.Edu. kjd.ucbvax.Berkeley.Edu. (
1 ; Serial
3600 ; Refresh
300 ; Retry
3600000 ; Expire
3600 ) ; Minimum
IN NS ucbvax.Berkeley.Edu.
1 IN PTR localhost.
```



## 5.5.5. Hosts

```

;
; @(#)ucb-hosts 1.1 (berkeley) 86/02/05
;
@      IN      SOA      ucbvax.Berkeley.Edu. kjd.monet.Berkeley.Edu. (
                                1.1      ; Serial
                                3600     ; Refresh
                                300      ; Retry
                                3600000  ; Expire
                                3600 )   ; Minimum
                                IN      NS      ucbarpa.Berkeley.Edu.
                                IN      NS      ucbvax.Berkeley.Edu.
localhost      IN      A      127.1
ucbarpa        IN      A      128.32.4
                                IN      A      10.0.0.78
                                ANY     HINFO    VAX-11/780 UNIX
arpa           IN      CNAME    ucbarpa
ernie          IN      A      128.32.6
                                ANY     HINFO    VAX-11/780 UNIX
ucbernie       IN      CNAME    ernie
monet          IN      A      128.32.7
                                IN      A      128.32.130.6
                                ANY     HINFO    VAX-11/750 UNIX
ucbmonet       IN      CNAME    monet
ucbvax         IN      A      10.2.0.78
                                IN      A      128.32.10
                                ANY     HINFO    VAX-11/750 UNIX
                                IN      WKS      128.32.0.10 UDP syslog route timed domain
                                IN      WKS      128.32.0.10 TCP ( echo telnet
                                discard sunrpc sftp
                                uucp-path systat daytime
                                netstat qotd nntp
                                link chargen ftp
                                auth time whois mtp
                                pop rje finger smtp
                                supdup hostnames
                                domain
                                nameserver )
vax            IN      CNAME    ucbvax
toybox         IN      A      128.32.131.119
                                ANY     HINFO    Pro350 RT11
toybox         IN      MX      0 monet.Berkeley.Edu
miriam         ANY     MB      vineyd.DEC.COM.
postmistress   ANY     MR      Miriam
Bind           ANY     MINFO    Bind-Request kjd.Berkeley.Edu.
                                ANY     MG      Ralph.Berkeley.Edu.
                                ANY     MG      Zhou.Berkeley.Edu.
                                ANY     MG      Painter.Berkeley.Edu.
                                ANY     MG      Riggle.Berkeley.Edu.
                                ANY     MG      Terry.pa.Xerox.Com.

```

## 5.5.6. host.rev

```

;
; @(#)ucb-hosts.rev 1.1 (Berkeley) 86/02/05
;
@      IN      SOA      ucbvax.Berkeley.Edu. kjd.monet.Berkeley.Edu. (
                                1.1      ; Serial
                                3600     ; Refresh
                                300      ; Retry
                                3600000  ; Expire
                                3600 ) ; Minimum
                                IN      NS      ucbarpa.Berkeley.Edu.
                                IN      NS      ucbvax.Berkeley.Edu.
4.0    IN      PTR      ucbarpa.Berkeley.Edu.
6.0    IN      PTR      ernie.Berkeley.Edu.
7.0    IN      PTR      monet.Berkeley.Edu.
10.0   IN      PTR      ucbvax.Berkeley.Edu.
6.130  IN      PTR      monet.Berkeley.Edu.

```



## 6. Domain Management

This section contains information for starting, controlling and debugging *named*.

## 6.1. /etc/rc.local

The hostname should be set to the full domain style name in */etc/rc.local* using *hostname(1)*. The following entry should be added to */etc/rc.local* to start up *named* at system boot time:

```

if [ -f /etc/named ]; then
    /etc/named [options] & echo -n ' named'      >/dev/console
fi

```

This usually directly follows the lines that start *syslogd*. Do Not attempt to run *named* from *inetd*. This will continuously restart the name server and defeat the purpose of having a cache.

## 6.2. /etc/named.pid

When *named* is successfully started up it writes its process id into the file */etc/named.pid*. This is useful to programs that want to send signals to *named*. The name of this file may be changed by defining *PIDFILE* to the new name when compiling *named*.

## 6.3. /etc/hosts

The *gethostbyname()* library call can detect if *named* is running. If it is determined that *named* is not running it will look in */etc/hosts* to resolve an address. This option was added to allow *ifconfig(8C)* to configure the machines local interfaces and to enable a system manager to access the network while the system is in single user mode.

It is advisable to put the local machines interface addresses and a couple of machine names and address in */etc/hosts* so the system manager can rcp files from another machine when the system is in single user mode. The format of */etc/hosts* has not changed. See *hosts(5)* for more information. Since the process of reading */etc/hosts* is slow, it is not advised to use this option when the system is in multi user mode.

#### 6.4. Signals

There are several signals that can be sent to the *named* process to have it do tasks without restarting the process.

##### 6.4.1. Reload

**SIGHUP** - Causes *named* to read *named.boot* and reload the database. All previously cached data is lost. This is useful when you have made a change to a data file and you want *named*'s internal database to reflect the change.

##### 6.4.2. Debugging

When *named* is running incorrectly, look first in */usr/adm/messages* and check for any messages logged by *syslog*. Next send it a signal to see what is happening.

**SIGINT** - Dumps the current data base and cache to */usr/tmp/named\_dump.db*. This should give you an indication to whether the data base was loaded correctly. The name of the dump file may be changed by defining *DUMPFIL* to the new name when compiling *named*.

*Note:* the following two signals only work when *named* is built with *DEBUG* defined.

**SIGUSR1** - Turns on debugging. Each following *USR1* increments the debug level. The output goes to */usr/tmp/named.run*. The name of this debug file may be changed by defining *DEBUGFILE* to the new name before compiling *named*.

**SIGUSR2** - Turns off debugging completely.

For more detailed debugging, define *DEBUG* when compiling the resolver routines into */lib/libc.a*.

### ACKNOWLEDGEMENTS

Many thanks to the users at U.C. Berkeley for falling into many of the holes involved with integrating BIND into the system so that others would be spared the trauma. I would also like to extend gratitude to Jim McGinness and Digital Equipment Corporation for permitting me to spend most of my time on this project.

Ralph Campbell, Doug Kingston, Craig Partridge, Smoot Carl-Mitchell, Mike Muuss and everyone else on the DARPA Internet who has contributed to the development of BIND. To the members of the original BIND project, Douglas Terry, Mark Painter, David Riggle and Songnian Zhou.

Anne Hughes, Jim Bloom and Kirk McKusick and the many others who have reviewed this paper giving considerable advice.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of

the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of Digital Equipment Corporation.



## REFERENCES

- [Birrell] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M.D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M.* 25, 4:260-274 April 1982.
- [RFC819] Su, Z. Postel, J., "The Domain Naming Convention for Internet User Applications." *Internet Request For Comment 819* Network Information Center, SRI International, Menlo Park, California. August 1982.
- [RFC882] Mockapetris, P., "Domain Names - Concept and Facilities." *Internet Request For Comment 882* Network Information Center, SRI International, Menlo Park, California. November 1983.
- [RFC883] Mockapetris, P., "Domain Names - Implementation and Specification." *Internet Request For Comment 883* Network Information Center, SRI International, Menlo Park, California. November 1983.
- [RFC973] Mockapetris, P., "Domain System Changes and Observations." *Internet Request For Comment 973* Network Information Center, SRI International, Menlo Park, California. February 1986.
- [RFC974] Partridge, C., "Mail Routing and The Domain System." *Internet Request For Comment 974* Network Information Center, SRI International, Menlo Park, California. February 1986.
- [Terry] Terry, D. B., Painter, M., Riggle, D. W., and Zhou, S., *The Berkeley Internet Name Domain Server*. Proceedings USENIX Summer Conference, Salt Lake City, Utah. June 1984, pages 23-31.
- [Zhou] Zhou, S., *The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers*. UCB/CSD 84/177. University of California, Berkeley, Computer Science Division. May 1984.



## Bug Fixes and Changes in 4.3BSD

April 15, 1986

*Marshall Kirk McKusick*

*James M. Bloom*

*Michael J. Karels*

Computer Systems Research Group  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California 94720  
(415) 642-7780

### ABSTRACT

This document briefly describes the changes in the Berkeley version of UNIX<sup>†</sup> for the VAX<sup>‡</sup> between the 4.2BSD distribution of July 1983 and this, its revision of March 1986. It attempts only to summarize the changes that have been made.

### Notable improvements

- The performance of the system has been improved to be at least as good as that of 4.1BSD, and in many instances is better. This was accomplished by improving the performance of kernel operations, rewriting C library routines for efficiency, and optimization of heavily used utilities.
- Many programs were rewritten to do I/O in optimal blocks for the filesystem. Most of these programs were doing their own I/O and not using the standard I/O library.
- The system now supports the Xerox Network System network communication protocols. Most of the remaining Internet dependencies in shared common code have been removed or generalized.
- The signal mechanism has been extended to allow selected signals to interrupt pending system calls.
- The C and Fortran 77 compilers have been modified so that they can generate single precision floating point operations.
- The Fortran 77 compiler and associated I/O library have undergone extensive changes to improve reliability and performance. Compilation may, optionally, include optimization phases to improve code density and decrease execution time. Many minor bugs in the C compiler have been fixed.
- The math library has been completely rewritten by a group of numerical analysts to improve both its speed and accuracy.
- Password lookup functions now use a hashed database rather than linear search of the password file.

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories.

<sup>‡</sup> DEC, VAX, PDP, MASSBUS, UNIBUS, Q-bus and ULTRIX are trademarks of Digital Equipment Corporation.



- C library string routines and several standard I/O functions were recoded in VAX assembler for greater speed. The C versions are available for portability. Standard error is now buffered within a single call to perform output.
- The symbolic debugger, *dbx*, has been dramatically improved. *Dbx* works on C, Pascal and Fortran 77 programs and allows users to set break points and trace execution by source code line numbers, references to memory locations, procedure entry, etc. *Dbx* allows users to reference structured and local variables using the program's programming language syntax.
- A new internet name domain server has been added to allow sites to administer their name space locally and export it to the rest of the Internet. Sites not using the name server may use a static host table with a hashed lookup mechanism.
- A new time synchronization server has been added to allow a set of machines to keep their clocks within tens of milliseconds of each other.

## Bug fixes and changes

### Section 1

adb	Locates the stack frame when debugging the kernel. Slight changes were made to output formats.
arcv	Has been retired to <i>/usr/old</i> .
as	The default data alignment may now be specified on the command line with a <i>-a</i> flag. A problem in handling filled data was fixed. Some bugs in the handling of <i>dbx</i> stab information were fixed.
at	The user may now choose to run <i>sh</i> or <i>csh</i> . Mail can now be sent to the user after the job has run; mail is always sent if there were any errors during execution. <i>At</i> now runs with the user's full permissions. All spool files are now owned by "daemon". The last update time is in seconds instead of hours. The problems with day and year increments have been fixed.
awk	Problems when writing to pipes have been corrected.
bc	<i>Bc</i> will continue reading from standard input, after failing to open a file specified from the command line.
calendar	Now allows tabs as separators. A subject line with the date of the reminder is added to each message.
cat	Problems opening standard input multiple times have been fixed. <i>Cat</i> now runs much faster in the default (optionless) case.
cb	No longer dumps core for unterminated comments or large block comments. For most purposes, <i>indent</i> (1) is far superior to <i>cb</i> .
cc	<p>The C compiler has some new features as well as numerous bug fixes. The principal new feature is a <i>-f</i> flag that tells the compiler to compute expressions of type <i>float</i> in single precision, following the ANSI C standard proposals. The C preprocessor has been extended to generate the dependency list for source files. The output is designed for inclusion in a makefile without modification.</p> <p>The bug fixes are many and varied. Several fixes deal with type coercion and sign extension. Signed char and short values are now properly sign-extended in comparisons with unsigned values of the same length. Conversion of a signed char value to unsigned short now correctly sign-extends to 16 bits (on the VAX). Non-integer switch expressions now elicit warnings and the appropriate conversions are emitted. Unsigned longs were being treated as signed for the purpose of conversion to floating types; the compiler now produces the appropriate complicated instruction sequence to</p>

do this right. An ancient misunderstanding that caused  $i * d$  to be treated as  $i = i * (\text{int}) d$  instead of  $i = (\text{double}) i * d$  for  $\text{int } i$  and  $\text{double } d$  has been corrected. If a signed integer division or modulus is cast to unsigned, the unsigned division or modulus routine is no longer used to compute the operation.

Some problems with bogus input and bogus output are now handled better; more syntax errors are caught and fewer code errors are emitted. Many declarations and expressions involving type `void` that used to be disallowed now work; some expressions that were not supposed to work are now caught. A pointer to a structure no longer stands a chance of being incremented by the size of its first element instead of the size of the structure when the value of the element is used at the same time the pointer is postincremented. Side effects in the left hand side of an unsigned assignment operator expression are now performed only once. Hex constants of the form `01234x56789` are now illegal. External declarations of functions may now possess arguments only if they are also definitions of functions. Declarations or initializations for objects of type `structure` where the particular structure was not previously defined used to result in confusing messages or even compiler errors; it's now possible to deduce one's mistake.

Some effort has been put into making the compiler more robust. Initializers containing casts sometimes would draw complaints about compiler loops or other problems; these now work properly. The register resource calculation now takes into account implicit conversions from `float` to `double` type, so that the code generator will not block by running out of registers. The compiler is more diligent about reducing structure type arguments to functions and no longer gives up when it cannot reduce the address to an offset from a register in only two tries. Programs that end in `"\n#"` no longer cause compiler core dumps. The compiler no longer dumps core for floating point exceptions that occur during reduction of constant expressions. The compiler expression tree table was enlarged so that it does not run out of space as quickly when processing complex expressions such as `putchar(c)`. The C preprocessor no longer uses a statically allocated space for strings. The preprocessor also now handles `#` line directives properly and correctly treats standard input from a terminal or a pipe. Two fencepost errors in the C peephole optimizer were adjusted and it now dumps core less often.

Some minor code efficiency changes were made. An important change is that the compiler now recognizes unsigned division and modulus operations that can be done with masking and shifting; this avoids the usual subroutine call overhead associated with these operations. The computation of register resources has improved so that the number of registers required for an expression is not overestimated as often. Register storage declarations for `float` variables now cause them to be put in registers if the `-f` flag is used. The compiler itself is somewhat faster, thanks primarily to a change that considerably reduces symbol table searches when entering and leaving blocks.

The compiler sources have been rearranged to make maintenance easier. The names of some source files have been changed to protect the innocent; header files now end in `.h`, and names of files reflect their functions. Configuration control has been simplified, so that only a simple configuration include file and the makefile flags variable should have to be considered when putting the compiler together. Redundant information has been eliminated from include files and the makefile, to reduce the chance of introducing changes that will make data structures or defines inconsistent. Values for opcodes are now taken from an include file `pcc.h` that is common to all the compilers that use the C compiler back end. The peephole optimizer can now be compiled without `-w`.

checknr  
chfn

The `.T& tbl` directive was added to the list of known commands.  
Has been merged into `passwd(1)`.



chgrp	An option has been added for recursively changing the group of a directory tree.
chmod	Can now recursively modify the permissions on a directory tree. The mode string was extended to turn on the execute bit conditionally if the file is executable or is a directory.
chsh	Has been merged into <i>passwd</i> (1).
clear	Now has a proper exit status.
colrm	Line length limitations have been removed.
compact	Has been retired to <i>/usr/old</i> .
compress	Replaces <i>compact</i> as the preferred method to use in saving file system space.
cp	No longer suffers problems when copying a directory to a nonexistent name or when some directories are not writable in a recursive copy. The <i>-p</i> flag was added to preserve modes and times when copying files.
crypt	Waits for <i>makekey</i> to finish before reading from its pipe.
csh	Has a new flag to stop argument processing so set user id shell scripts are more secure. File name completion may be optionally enabled. <i>Csh</i> keeps better track of the current directory when traversing symbolic links. Some major work was done on performance.
ctags	<i>Ctags</i> was modified to recognize LEX and YACC input files. Files ending in <i>.y</i> are presumed to be YACC input, and a tag is generated for each non-terminal defined, plus a tag <i>yyvsparse</i> for the first <i>%%</i> line in the file. Files ending in <i>.l</i> are checked to see if they are LEX or Lisp files. A tag <i>yylex</i> is generated for the first <i>%%</i> line in a LEX file. In addition, for both kinds of files, any C source after a second <i>%%</i> is scanned for tags.
date	The <i>date</i> command can now be used to set the date on all machines in a network using the <i>timed</i> (8) program. More information is logged regarding the setting of time.
dbx	Major improvements have been made to <i>dbx</i> since the 4.2BSD release. Large numbers of bug fixes have made <i>dbx</i> much more pleasant to use; in particular many pointer errors that used to cause <i>dbx</i> to crash have been caught. Some new features have been installed; for instance it is now possible to search for source lines with regular expressions. The Fortran and Pascal language support is much improved, and the DEC Western Research Labs Modula-2 compiler is now supported.
dd	Exit codes have been changed to correspond with normal conventions.
deroff	<i>Deroff</i> no longer throws out two letter words.
diff	Context diffs merge nearby changes. New flags were added for ignoring white space differences and for insensitivity to case.
diff3	The RCS version of <i>diff3</i> has been merged into the standard <i>diff3</i> under two new flags, <i>-E</i> and <i>-X</i> .
echo	No longer accepts <i>-anything</i> in place of <i>-n</i> .
error	Support for the DEC Western Research Labs Modula-2 compiler has been added. <i>Error</i> will now be able to run when there is no associated tty, so it may now be driven from <i>at</i> (1), etc. If the <i>-n</i> and <i>-t</i> options are selected, <i>error</i> will not touch files.
ex	Support for changing window size has been added, and terminals with many lines, such as the WE5620, are now handled. Several small bug fixes were installed and various facilities have been made faster. <i>Ex</i> only reads the file <i>.exrc</i> if it is owned by the user, unless the <i>sourceany</i> option is set. It only looks for "mode lines" if the <i>modeline</i> option is set. If Lisp mode is set, it allows "-" to be used in "words". <i>Expreserve</i> now provides a better description of what happened to a user's buffer when disaster struck.

eyacc

*eyacc* is no longer a standard utility. It has been moved to the Pascal source directory.

f77

The Fortran compiler has been substantially improved. Many serious bugs have been fixed since the last release; the compiler now passes several widely used tests such as the Navy Fortran Compiler Validation System and the IMSL and NAG mathematical libraries. The optimizer is now trustworthy and robust; the many gruesome bugs that it used to inflict on programs, such as resolving different variables in the same **common** block into the same temporary for purposes of common subexpression elimination, have been fixed. **Do** loops, which used to suffer from deadly problems where loop variables, limit values and tests all managed to misfire even without the help of the optimizer, now produce proper results. Many severe bugs with character variables and expressions have been fixed; it is now possible to have variable length character variables on either side of an assignment, and the lengths of concatenations are properly computed. Several register allocation bugs have been fixed, among them the awful bug that  $a = f(a)$  where  $a$  is in a register would not alter the value of  $a$ . Register allocation, though significantly improved, is still pitifully naive compared with the methods found in production Fortran compilers. **Save** statements cause variables to be retained, even if a subroutine returns from inside a loop. It is no longer possible to modify constants that are passed as parameters to subroutines and thus change all future uses of the constant when it is used as a subroutine parameter. Multi-level equivalences are no longer scrambled, and the **cmplx** intrinsic conversion function no longer garbles its result. The compiler now generates integer move instructions where it used to produce floating point move instructions, even when not optimizing, so that non-standard use of equivalences between real and integer types work as on most other systems. **Assign** statements now work with **format** statements. The "first character" parameter of a substring is now evaluated only once instead of twice. Restrictions on parameter variables are now enforced, and the compiler no longer aborts while trying to make sense of impossible parameter variables. The restrictions on array dimension declarators are much closer to the standard and much more stringent. Statement ordering used to be much more flexible, and wrong; it is now strictly enforced, leading to fewer compiler errors. The compiler now chides the user for declaring adjustable length character variables that are not dummy arguments. The compiler understands that subroutines and functions are different and prevents them from being used interchangeably. The parser is no longer fooled by excess "positional I/O control" parameters in I/O statements.

Several changes have been made to prevent the compiler itself from aborting; in particular, computed **gotos** do not elicit compiler core dumps, nor do multiplications by zero, nor do unusual statement numbers. The compiler now recognizes and complains about various kinds of hardware errors that can result from evaluating constant expressions, such as integer and floating overflow; it no longer dies when it receives a SIGFPE. Several memory management bugs that caused the compiler to dump core for seemingly random things have met their demise. Some conversion operations used to cause the code generator to emit impossible assembly language instructions that in turn caused the assembler some indigestion; these are now fixed. Some symbol table modifications were made to help out *dbx*(1), so that values of **common** and **parameter** storage classes and logical types are now accessible from *dbx*. When the compiler does abort, the error messages produced are now comprehensible to human beings and messy core dumps are no longer left behind. Some effort has been made to improve error reporting for program errors and to handle exceptional conditions in which the old compiler used to punt.

Some improvements in optimization were added to the compiler. Offsets to static data are now shorter than before; the compiler used to produce 32-bit offsets for all local variables. Real variables may now be allocated to registers. Format strings in **format** statements are compiled for considerable runtime savings; for various reasons, format strings in character constants and variables in I/O statements are not.

Common subexpression elimination now reduces the re-evaluation of exponentiations in polynomial expressions. Some problems with alignment of data that caused ghastly performance degradation have been repaired.

Some changes have been made in the way the compiler is put together. The compiler front end now uses the common intermediate code format established in the include file *pcc.h* to communicate with the back end. The back end has been re-merged with the C compiler sources, so that bug fixes to the C compiler are automatically propagated to the Fortran back end. Similarly, the Fortran and C peephole optimizers were re-merged.

Some new features were added to the compiler. There is now a *-r8* flag to coerce real and complex variables and constants to double precision and double complex types for extended precision. There is a *-q* flag to suppress listing of file and entry names during compilation. Some foolproofing was added to the compiler driver; it is no longer possible to wipe out a source file by entering "f77 -o foo.f", and it now complains about incompatible combinations of options.

Many I/O library bugs were fixed. Auxiliary I/O has been fixed to be closer to the standard: *close* is a no-op on a non-existent or unconnected unit; *rewind* and *backspace* are no-ops on an unconnected unit; *endfile* opens an unconnected unit. *Inquire* returns true when asked if units 0-MAXUNIT exist, false for other integers; it used to return false for legal but unconnected file numbers and errors for illegal numbers. *Inquire* now fills in all requested fields, even if the file or unit does not exist or is unconnected. *Inquire* by unit now correctly returns the unit number. Most of the formatted I/O input scanning has been rewritten to check for invalid input. For example, with an *f10.0* format term, the following all used to read as 12.345: "1+2.345", "12.3abc45", "12.3.45", "12345e1.-"; they now generate errors. Conversely, the legal datum "12345-2" for 12.345 used to be misread as -1234.52. The *b* format term is now fixed, and *bz* now works for short records. Reads of short logical variables no longer overwrite neighboring data in memory. Infinite loops in formatted output (an I/O list but no conversion terms in the format) are now caught, printing multiple records after the list is exhausted. In list directed reads, a repeat count, *r*, followed by an asterisk and a space (and no comma) now follows the standard and skips *r* list items. Repeat counts for complex constants now work. Tabs are now fully equivalent to spaces in list directed input. There are two new formatting terms, *x* for hex and *o* for octal. The library now attempts to get to the next record if doing an *err=* branch on error; the standard does not require this, but it is undesirable to leave the system hanging in mid record. After input errors, the I/O library now tries to skip to the next line if there is another read. This functionality is not required by the standard and is still not guaranteed to work.

The Fortran runtime and I/O libraries have several new features. Many routines and variables have been made static, cutting the number of symbols defined by the library almost in half. Many source files have been reorganized to eliminate the loading of extraneous routines; for example, the formatted read routines are not loaded if a program only performs formatted writes. Standard error is now buffered. All error processing is now centralized in a single routine, *f77\_abort*. The *f77\_abort* routine has been separated from the normal Fortran main routine so that C code can call Fortran subroutines. Fortran programs that abort normally get a core file only if they are loaded with *-g*; the environment variable *f77\_dump\_flag* may be used to override this by setting it to *y* or *n*. The *rindex* routine now works as documented. The C library *malloc* and *random* routines may now be accessed from Fortran.

The new VAX math library has been incorporated and some bugs in calling math library routines have been fixed. The routine *d\_dprod* was added for use with the *-r8* flag. The *sinh* and *tanh* routines have been deleted as they are loaded directly from the math library. The *log10* routine from the math library is now used by *r\_log10* and



*d\_lg10*. The *pow* routines now divide by zero when zero is raised to a negative power so as to generate an exception. Complex division by zero now generates an error message.

Appropriately named environment variables now override default file names and names in open statements; see "Introduction to the f77 I/O Library" for details. Unit numbers may vary from 0 to 99; the maximum number that can be open simultaneously depends on the system configuration limit (the library does not check this value). Namelist I/O similar to that in VMS Fortran has been added to the compiler, and library routines to implement it have been added to the I/O library. The documents "A Portable Fortran 77 Compiler" and "Introduction to the f77 I/O Library" have been revised to describe these changes. The new *help* system on the distribution tape in the user contributed software section contains a large set of help files for f77.

fed	Has been retired to <i>/usr/old</i> .
find	Some new options have been added. It is now possible to choose users or groups that have no names by using the <i>-nouser</i> and <i>-nogroup</i> options. The <i>-ls</i> option provides a built in <i>ls</i> facility to allow the printing of various file attributes; it is identical to " <i>ls -lgids</i> ". It is now possible to restrict <i>find</i> to the file system of the initial path name with the <i>-xdev</i> option. A new type, <i>-type s</i> , for sockets has been added. Symbolic links are now handled better. Globbing is now faster. <i>Find</i> supports an abbreviated notation, " <i>find pattern</i> ," which searches for a pattern in a database of the system's path names; this is much faster than the standard method.
finger	Despite numerous changes, <i>finger</i> still has Berkeley parochialisms. It has been modified to provide finger information over the network. Control characters are mapped to their printable equivalents (e.g. ^X) to avoid trojan horses in <i>.plan</i> and <i>.profile</i> files.
file	<i>File</i> has been extended to recognize sockets, compressed files ( <i>.Z</i> ), and shell scripts. When it determines that a file is a shell script, it tries to discover whether it is a Bourne shell script or a C shell script. The special bits set user id, sticky, and append-only are also noted. The value of a symbolic link is now printed.
from	An error message is printed if the requested mailbox cannot be opened.
ftp	Many bugs have been fixed. New features are: support for new RFC959 FTP features (such as "store unique"), new commands that manipulate local and remote file names to better support connections to non-UNIX systems, support for third party file transfers between two simultaneously connected remote hosts, transfer abort support, expanded and documented initialization procedures (the <i>.netrc</i> file), and a simple command macro facility.
gprof	Uses <i>setitimer</i> to discover the clock frequency instead of looking it up in <i>/dev/kmem</i> . An alphabetical index printing routine has been added. A few changes were made to the output format; a new column indicates milliseconds per call.
groups	Now prints out the group listed in the password file in addition to the groups listed in the groups file.
help	Has been superseded by the <i>help</i> facility included in the User Contributed Software.
hostid	Has been extended to take an Internet address or hostname.
indent	Has been completely rewritten; its default mode now produces programs somewhat more closely reflecting the local Berkeley style.
install	The <i>chmod</i> in the <i>install</i> script uses <i>-f</i> so that it does not complain if it fails. When <i>mv'ing</i> and <i>strip'ing</i> a binary ( <i>-s</i> and not <i>-c</i> ), the <i>strip</i> is done before the <i>mv</i> to avoid fragmentation on the destination file system.
iostat	Disk statistics are collected by an alternate clock, if it exists. Overflow detection has been added to avoid printing negative times. A call to <i>flush</i> was added so that <i>iostat</i>

	works through pipes and sockets. Code to handle additional disks was added in the same way as in <i>vmstat</i> . The header is reprinted when <i>lastat</i> is restarted.
<b>kill</b>	Signal 0 may now be used as documented.
<b>lastcomm</b>	Several bug fixes were installed. <i>Lastcomm</i> now understands the revised accounting units.
<b>ld</b>	A list of directories to search for libraries may now be specified on the command line.
<b>learn</b>	The "files" lesson has been updated to reflect the default system tty conventions for erase and kill characters. <i>Learn</i> now uses directory access routines so that trash files can be removed properly between lessons.
<b>leave</b>	Now ignores SIGTTOU and properly handles the <i>+hmm</i> option.
<b>lex</b>	The error messages have been made more informative.
<b>lint</b>	Tests for negative or excessively large constant shifts were added. For <i>-a</i> , warnings for expressions of type <i>long</i> that are cast to type <i>void</i> are no longer emitted. A bug which caused <i>lint</i> to incorrectly report clashes for the return types of functions has been fixed. <i>Lint</i> now understands that enums are not ints. The lint description for the C library was updated to reflect sections two and three of the Programmers Manual more accurately. Several more libraries in <i>/usr/lib</i> now have lint libraries. Changes were made to accommodate the restructuring of the C compiler for common header files.
<b>lisp</b>	The Berkeley version of Franz Lisp has not been changed much since the 4.2BSD release. It has been updated to reflect changes in the C library.
<b>ln</b>	Now prints a more accurate error message when asked to make a symbolic link into an unwritable directory.
<b>lock</b>	<i>Lock</i> now has a default fifteen minute timeout. The root password may be used to override the lock. If an EOF is typed, it is now cleared instead of spinning in a tight loop until the timeout period.
<b>logger</b>	A new program that logs its standard input using <i>syslog(3)</i> .
<b>login</b>	The environment may be set up by another process that calls <i>login</i> . It now uses the new <i>gettyent(3)</i> routines to read <i>/etc/ttys</i> .
<b>lpr</b>	Now supports "restricted access" to a printer- printer use may be restricted to only those users in a specific group-id.
<b>mail</b>	<i>Mail</i> now expects RFC822 headers instead of the obsolete RFC733 headers. A <i>retain</i> command has been added. If the <i>PAGER</i> variable is set in the environment, it is used to page messages instead of <i>more(1)</i> . The <i>write</i> command now deletes the entire header instead of only the first line. An <i>unread/Unread</i> command (to mark messages as not read) was added. If <i>Replyall</i> is set, the senses of <i>reply</i> and <i>Reply</i> are reversed. When editing a different file, <i>mail</i> always prints the headers of the first few messages. <i>Flock(2)</i> is used for mailbox locking. Commands <i>"-"</i> and <i>"+"</i> skip over deleted messages; type <i>user</i> now does a substring match instead of a literal comparison. A <i>-I</i> flag was added which causes <i>mail</i> to assume that input is a terminal.
<b>make</b>	A bug which caused <i>make</i> to run out of file descriptors because too many files and directories were left open has been fixed. Long path names should not be a problem now. A <i>VPATH</i> macro has been added to allow the user to specify a path of directories to search for source files.
<b>man</b>	Support for alternate manual directories for <i>man</i> , <i>apropos</i> and <i>whatis</i> was added. A side effect of this is that the <i>whatis</i> database was moved to the <i>man</i> directory. If the source for a manual page is not available, <i>man</i> will display the formatted version. This allows machines to avoid storing both formatted and unformatted versions of the manual pages. The environment variable <i>MANPATH</i> overrides the default directory <i>/usr/man</i> . The <i>-t</i> option is no longer supported. The printing process has been



	streamlined by using "more -s catfile" instead of "cat -s catfile   ul   more -f". Searches of <i>/usr/man/mano</i> are more lenient about file name extensions. The source for <i>man</i> was considerably cleaned up; the magic search lists and commands were put at the top of the source file and the private copy of <i>system</i> was deleted.
<b>mesg</b>	So that terminals need not be writable to the world, <i>mesg</i> only changes the group "write" permission. (Terminals are now placed in group <i>tty</i> so that users may restrict terminal write permission to programs which are set-group-id <i>tty</i> .)
<b>mkdir</b>	Prints a "usage" error message instead of an uninformative "arg count" message.
<b>more</b>	Now allows backward scanning. It will also handle window size changes. It simulates "crt" style erase and kill processing if the terminal mode includes those options.
<b>msgs</b>	Will no longer update <i>msgsrc</i> if the saved message number is out of bounds.
<b>mv</b>	No longer runs <i>cp(1)</i> to copy a file; instead it does the copy itself.
<b>netstat</b>	Routes and interfaces for Xerox NS networks are now shown. The -I option has been added to specify a particular interface for the default display. The -u option has been added to show UNIX domain information. Several new mbuf types and statistics are now displayed; subnetting is now understood.
<b>nice</b>	Is relative as documented, not absolute.
<b>nroff</b>	No longer replaces single spaces with tabs when using the -h option.
<b>Pascal</b>	The Pascal compiler and interpreter have been extensively rewritten so that they will (nearly) pass through <i>lint</i> . In theory they have not changed from a semantic point of view. A few bugs have been fixed, and undoubtedly some new ones introduced. The Pascal runtime support has improved error diagnostics. Real number input scanning now corresponds to standard Pascal conventions rather than those of <i>scanf(3S)</i> .
<b>passwd</b>	The <i>passwd</i> program incorporates the functions of <i>chfn</i> and <i>chsh</i> under -f and -s flags. Whenever information is changed <i>passwd</i> also updates the associated <i>ndbm(3X)</i> database used by <i>getpwnam</i> and <i>getpwuid</i> . Office room and phone numbers are less dependent on Berkeley's usage. Checks are made for write errors before renaming the password file.
<b>plot</b>	The output device resolution can now be specified using the -r option. Support has been added for the Imagen laser printer and the Tektronix 4013.
<b>pr</b>	The buffer is now large enough for 66 x 132 output.
<b>print</b>	Has been retired to <i>/usr/old</i> ; use "lpr -p" instead.
<b>prmail</b>	Has been retired to <i>/usr/old</i> ; use "Mail -u user" instead.
<b>prof</b>	Uses <i>setitimer</i> to determine the clock frequency instead of assuming 60 hertz.
<b>ps</b>	Saves static information for faster startup. It now <i>l</i> ints symbolic values for wait channels.
<b>pti</b>	Has been retired to <i>/usr/old</i> .
<b>ptx</b>	Cleans up after itself and exits with a zero status on successful completion.
<b>quota</b>	Verifies that the system supports quotas before trying to interpret the quota files.
<b>ranlib</b>	The -t option updates a library's internal time stamp without rebuilding the table of contents. "Old format" and "mangled string table" are now warnings rather than fatal errors. Memory allocation is done dynamically.
<b>rcp</b>	For the convenience of system managers, <i>rcp</i> has moved from <i>/usr/ucb</i> to <i>/bin</i> , hence it can be used without mounting <i>/usr</i> . Remote user names are now specified as <i>user@host</i> instead of <i>host.user</i> to support Internet domain hostnames that contain periods ("."). A -p option has been added that preserves file and directory modes, access time, and modify time. It now uses <i>getservbyname</i> instead of compile time constants.



<b>rdist</b>	A new program that keeps files on multiple machines consistent with those on a master machine.
<b>refer</b>	The key letter code was fixed so that control characters are not generated. Several problems that caused the generation of duplicate citations, particularly with the <code>-e</code> and <code>-s</code> options, have been fixed. EOF on standard input is now properly handled. <i>Refer</i> folds upper and lower case when sorting.
<b>rlogin</b>	<i>Rlogin</i> negotiates with <i>rlogind</i> to determine whether window size changes should be passed through. If the remote end is running a 4.3BSD <i>rlogind</i> , it will agree to accept and pass through SIGWINCH signals to user processes under its control. The <code>-8</code> flag allows an 8-bit path on input. The <code>-L</code> flag allows an 8-bit path on output. The escape character is now echoed as soon as a second non-command character is typed. A new command character <code>^Y</code> has been added to suspend only the input end of the session without stopping output from the remote end (unless <code>tostop</code> has been set). The <i>ioctl</i> TIOCSGRP has been changed to <i>fcntl</i> F_SETOWN. Several changes have been made to reduce the amount of data sent after an interrupt has been typed, and to avoid flushing data when changing modes.
<b>rm</b>	The <code>-f</code> option produces no error messages and exits with status 0. The problem of running out of file descriptors when doing a recursive remove have been fixed.
<b>rmdir</b>	Improved error messages, in the same fashion as <i>mkdir</i> .
<b>rsh</b>	The <code>-L</code> , <code>-w</code> , and <code>-8</code> flags are ignored so that they may be passed along with <code>-e</code> to <i>rlogin</i> .
<b>ruptime</b>	The <code>-r</code> flag has been added to reverse sort order.
<b>rwho</b>	Now allows hosts with long names (greater than 16 characters).
<b>script</b>	Now propagates window size changes.
<b>sed</b>	No longer loops when the first regular expression is null.
<b>sendbug</b>	Allows command line <code>-D</code> arguments to override built in defaults for name and host address of the bugs mailing list. The "Repeat-By" field is now optional. <i>Sendbug</i> now checks the EDITOR environment variable instead of assuming <i>vi</i> .
<b>sh</b>	"#" is no longer considered a comment character when <i>sh</i> is interactive. The IFS variable is not imported when <i>sh</i> runs as root or if the effective user id differs from the real user id.
<b>size</b>	Now exits with the number of errors encountered.
<b>sort</b>	Checks for and exits on write errors.
<b>spell</b>	A couple of trouble-causing words have been removed from <i>spell</i> 's stoplist; e.g. "reus" that caused "reused" to be flagged. A few words that <i>spell</i> would not derive have been removed from the stoplist. Several hundred words that <i>spell</i> derives without difficulty from existing words (e.g. "getting" from "get"), or that <i>spell</i> would accept anyway, e.g. "1st, 2nd" etc., have been removed from <i>/usr/dict/words</i> .
<b>stty</b>	Has been extended to handle window sizes and 8-bit input data paths. "stty size" prints only the size of the associated terminal.
<b>su</b>	Only members of group 0 may become root.
<b>symorder</b>	Now reorders the string table as well as the name list.
<b>sysline</b>	Now understands how to run in one-line windows and how to adjust to window size changes. Numerous small changes have been made in the output format.
<b>systat</b>	A new program that provides a cursed form of <i>vmstat</i> , as well as several other status displays.
<b>tail</b>	Makes use of a much larger buffer.

<b>talk</b>	The new version of <i>talk</i> has an incompatible but well-defined protocol that works across a much broader range of architectures. The new talk rendezvous at a new port so that the old version can still be used during the conversion. <i>Talkd</i> looks for a writable terminal instead of giving up if a user's first entry in <i>/etc/utmp</i> is not writable. Root may always interrupt. <i>Talk</i> now runs <i>set-group-id</i> to group <i>tty</i> so that it is no longer necessary to make terminals world writable.
<b>tar</b>	Preserves modified times of extracted directories. The <b>-B</b> option is turned on when reading from standard input. Some sections were rewritten for efficiency.
<b>tbl</b>	The hardwired line length has been removed.
<b>tcopy</b>	A new program for doing tape to tape copy of multifile, arbitrarily blocked magnetic tapes.
<b>tee</b>	<i>Tee's</i> buffer size was increased.
<b>telnet</b>	<i>Telnet</i> first tries to interpret the destination as an address; if that fails, it is then passed off to <i>gethostbyname</i> . If multiple addresses are returned, each is tried in turn until one succeeds, or the list is exhausted. If a non-standard port is specified, the initial "Suppress Go Ahead" option is not sent. Commands were added to escape the escape character, send an interrupt command, and send "Are You There". Carriage return is now mapped to carriage return, newline.
<b>tftp</b>	Has many bug fixes. It no longer loops upon reading EOF from standard input. Retransmission to send was added, as well as an input buffer flush to both send and receive.
<b>tip</b>	Lock files are no longer left lying about after <i>tip</i> exits, and the <i>uucp</i> spool directory does not need to be world writable. A new "\$" command sends output from a local program to a remote host. Alternate phone numbers are separated only by ",", thus several dialer characters that were previously illegal may now be used. <i>Tip</i> now arranges to copy a phone number argument to a safe place, then zero out the original version. This narrows the window in which the phone number is visible to miscreants using <i>ps</i> or <i>w</i> . Also fixed was a bug that caused the phone number to be written in place of the connection message. Carrier loss is recognized and an appropriate disconnect action is taken. Bugs in calculating time and fielding signals have been fixed. Several new dialers were added.
<b>tn3270</b>	A new program for emulating an IBM 3270 over a <i>telnet</i> connection.
<b>tp</b>	Memory allocation was changed to avoid <i>realloc</i> .
<b>tr</b>	Checks for and exits on write errors.
<b>trman</b>	Has been retired to <i>/usr/old</i> .
<b>tset</b>	Can now set the interrupt character. The defaults have been changed when the interrupt, kill, or erase characters are NULL. <i>Reset</i> is now part of <i>tset</i> . The window size is set if it has not already been set. <i>Tset</i> continues to prompt as long as the terminal type is unknown.
<b>users</b>	Now much quieter if there are no users logged on.
<b>uucp</b>	Several fixes and changes from the Usenet have been incorporated. The maximum length of a sitename has been increased from 7 to 14 characters. <i>Uucp</i> has been changed to understand the new format of <i>/etc/ttys</i> . Support for more dialers has been added.
<b>vacation</b>	A new program that answers mail while you are on vacation.
<b>vgrind</b>	Has been extended to handle the DEC Western Research Labs Modula-2 compiler and <i>yacc</i> .
<b>vlp</b>	Now properly handles indented lines.

vmstat	The -i flag was added to summarize interrupt activity. The -s listing was expanded to include cache hit rates for the name cache and the text cache. The standard display has been generalized to allow command line selection of the disks to be displayed. A new header is printed after the program is restarted. If an alternative clock is being used to gather statistics, it is properly taken into account.
vpr	Has been retired to <i>/usr/old</i> .
w	Users logged in for more than one day have login day and hour listed; users idle for more than one day have their idle time listed in days.
wall	Will now notify all users on large systems.
whereis	Now also checks <i>man1</i> , <i>mann</i> , and <i>mano</i> .
which	Now sets prompt before sourcing the user's <i>.cshrc</i> file to ensure that initialization for interactive shells is done.
whoami	Uses the effective user id instead of the real user id.
window	A new program that provides multiple windows on ASCII terminals.
write	Looks for a writable terminal instead of giving up if a user's first entry in <i>/etc/utmp</i> is not writable. Root may always interrupt. Non-printable escape sequences can no longer be sent to an unsuspecting user's terminal. <i>Write</i> now runs <i>set-group-id</i> to group <i>tty</i> so that it is no longer necessary to make terminals world writable.
xsend	Notice of secret mail is now sent with a subject line showing who sent the mail. The body of the message includes the name of the machine on which the mail can be read.
xstr	Now handles multiple-line strings.

## Section 2

The error codes for Section 2 entries have been carefully scrutinized to insure that the documentation properly reflects the source code. User-visible changes in this section lie mostly in the area of the interprocess communication facilities; the Xerox Network System communication protocols have been added and the existing communication facilities have been extended and made more robust.

adjtime	A new system call which skews the system clock to correct the time of day.
fcntl	The FASYNC option to enable the SIGIO signal now works with sockets as well as with ttys. The interpretation of process groups set with F_SETOWN is the same for sockets and for ttys; negative values refer to process groups, positive values to processes. This is the reverse of the previous interpretation of socket process groups set using <i>ioctl</i> to enable SIGURG.
kill	The error returned when trying to signal one's own process group when no process group is set was changed to ESRCH. Signal 0 can now be used as documented.
lseek	Returns an EPIPE error when seeking on sockets (including pipes) for backward compatibility.
open	When doing an open with flags O_CREAT and O_EXCL (create only if the file did not exist), it is now considered to be an error if the target exists and is a symbolic link, even if the symbolic link refers to a nonexistent file. This behavior was added for the security of programs that need to create files with predictable names.
ptrace	A new header file, <i>&lt;sys/ptrace.h&gt;</i> , defines the request types. When the process being traced stops, the parent now receives a SIGCHLD.
readlink	Returns EINVAL instead of ENXIO when trying to read something other than a symbolic link.

<b>rename</b>	If the ISVTX (sticky text) bit is set in the mode of a directory, files in that directory may not be the source or target of a <i>rename</i> except by the owner of the file, the owner of the directory, or the superuser.
<b>select</b>	Now handles more descriptors. The mask arguments to <i>select</i> are now treated as pointers to arrays of integers, with the first argument determining the size of the array. A set of macros in <i>&lt;sys/types.h&gt;</i> is provided for manipulating the file descriptor sets. The descriptor masks are only modified when no error is returned.
<b>setsockopt</b>	Options that could only be <i>set</i> in 4.2BSD (e.g. SO_DEBUG, SO_REUSEADDR) can now be set or reset. To implement this change all options must now supply an option value which specifies if the option is to be turned on or off. The SO_LINGER option takes a structure as its option value, including both a boolean and an interval. New options have been added: to get or set the amount of buffering allocated for the socket, to get the type of the socket, and to check on error status. Options can be set in any protocol layer that supports them; IP, TCP and SPP all use this mechanism.
<b>setpriority</b>	The error returned on an attempt to change another user's priority was changed from EACCES to EPERM.
<b>setreuid</b>	Now sets the process <i>p_uid</i> to the new effective user ID instead of the real ID for consistency with usage elsewhere. This avoids problems with processes that are not able to signal themselves.
<b>sigreturn</b>	Is a new system call designed for restoring a process' context to a previously saved one (see <i>setjmp/longjmp</i> ).
<b>sigvec</b>	Three new signals have been added, SIGWINCH, SIGUSR1, and SIGUSR2. The first is for notification of window size changes and the other two have been reserved for users.
<b>socket</b>	The usage of the (undocumented) SIOCSPGRP <i>ioctl</i> has changed. For consistency with <i>fcntl</i> , the argument is treated as a process if positive and as a process group if negative. Asynchronous I/O using SIGIO is now possible on sockets.
<b>swapon</b>	The error returned for when requesting a device which was not configured as a swap device was changed from ENODEV to EINVAL. In addition, <i>swapon</i> now searches the swap device tables from the beginning instead of the second entry.
<b>unlink</b>	If the ISVTX (sticky text) bit is set in the mode of a directory, files may only be removed from that directory by the owner of the file, the owner of the directory, or the superuser.

### Section 3

The Section 3 documentation has been reorganized into just two sections. The first section contains everything previously in Section 3 except the Fortran library routines. The second section contains the Fortran library routines.

The routines *memccpy*, *memchr*, *memcmp*, *memcpy*, *memset*, *strchr*, *strcspn*, *strpbrk*, *strchr*, *strspn*, and *strtok* have been added for compatibility with System V. These routines are similar to the string and block handling ones described in the *bstring* and *string* manual pages. The 4.3BSD *string* and *bstring* versions should be faster than these compatibility routines on the VAX.

<b>abort</b>	Sets SIGILL signal action to the default to avoid looping if SIGILL had been ignored or blocked.
<b>ctime</b>	Daylight savings time calculations have been fixed for Europe and Canada. Programs making multiple calls to <i>ctime</i> will make fewer system calls. The include file has moved from <i>&lt;sys/time.h&gt;</i> to <i>&lt;time.h&gt;</i> .
<b>ctype</b>	<i>iscntrl</i> has been fixed to correspond to the manual page. Space is a printing character. <i>isgraph</i> is a new function that returns true for characters that leave a mark on the

	paper. <i>toupper</i> , <i>tolower</i> , and <i>toascii</i> have all been documented.
curses	The library handles larger termcap definitions and handles more of the “funny” termcap capabilities. The old <i>crmode</i> and <i>nocrmode</i> macros have been renamed <i>cbreak</i> and <i>nocbreak</i> respectively; backwards compatible definitions for these macros are provided. The erase and kill characters and the terminal’s baudrate may be accessed via <i>erasechar</i> , <i>killchar</i> , and <i>baudrate</i> macros defined in <i>&lt; curses.h&gt;</i> . A <i>touchoverlap</i> function has been provided, and bugs in <i>overlay</i> and <i>overwrite</i> have been fixed.
dbm	Has been rewritten to use the multiple-database version of the library, <i>ndbm</i> .
disktab	Has added support for two new fields indicating the use of <i>bad144</i> -style bad sector forwarding and filesystem offsets specified in sectors.
encrypt	Now works correctly when called directly.
execvp	No longer recognizes “.” as a path separator.
frexp	Now handles 0 and powers of 2 correctly. This routine is now written in assembly language for the VAX.
gethost*	<i>gethostbyaddr</i> and <i>gethostbyname</i> have been modified to make calls to the name server. If the name server is not running, a linear scan of the host table is made. With an optional C library configuration, these routines may instead use an <i>ndbm</i> database for the host table. One of these lookup mechanisms must be specified when compiling the C library. The default is to use the name server. <i>gethostent</i> has no equivalent when using the routines calling the name server. The <i>hostent</i> structure has been modified to support the return of multiple addresses. The external variable <i>h_errno</i> has been added for returning error status information from the name server, such as whether a transient error was encountered.
getopt	A new routine for parsing command line arguments. It is compatible with the System V routine by the same name.
getpw*	<i>getpwnam</i> and <i>getpwuid</i> use a hashed database using <i>ndbm</i> for faster lookups by user name and id.
getty*	<i>gettyent</i> and <i>gettynam</i> are new routines for looking up entries in the new version of <i>/etc/tty</i> s. The new header file <i>&lt; ttyent.h&gt;</i> describes the associated structures.
getusershell	A new routine for retrieving shell names from a file listing the standard interactive shells, <i>/etc/shells</i> , for the use of <i>passwd</i> (1) and servers providing remote host access.
getwd	<i>Getwd</i> no longer changes directories in calculating the working directory; this eliminates problems with return to the current directory, and results in fewer <i>stat</i> calls.
inet_makeaddr	Properly handles INADDR_BROADCAST.
longjmp	On errors, <i>longjmp</i> calls the routine <i>longjmperror</i> . The default routine still prints “longjmp botch” and exits; this may be replaced if a program wants to provide its own error handler.
malloc	<i>Malloc</i> underwent a major rework. Memory requests of page size or larger are always page aligned, and are now optimized for sizes that are a power of two. The debugging code has been improved.
math	The math library has been rewritten to improve the speed and accuracy of the routines on VAXen with D-format floating point support and machines that conform to the IEEE standard 754 for double precision floating point arithmetic. The library also has improved error detection and handling; for the VAX, the library generates reserved operand faults for invalid operands. Many new functions have been added. Two functions have changed their names; <i>gamma</i> is now <i>lgamma</i> and <i>fmod</i> is now <i>modf</i> . The old math library is available as <i>-lom</i> .
mkstemp	Is a new routine similar to <i>mktemp</i> except that it returns an open file descriptor for a temporary file. It is intended to replace <i>mktemp</i> in programs (run as root or setuid)

	that must be concerned with atomic creation of temporary files without the possibility of having the temporary file relocated to an unexpected location by a symbolic link.
<b>ndbm</b>	A new version of <i>dbm</i> that allows multiple databases to be open simultaneously.
<b>nlist</b>	Now returns -1 on error or the number of unfound items.
<b>perror</b>	A few of the error messages have been made more accurate.
<b>plot</b>	Supports many new devices: Tektronix 4013, AED graphics terminal, BBN Bitgraph terminal, terminals using the DEC GiGi protocol, HP 2648 terminals and 7221 plotters, and Imagen laser printers (240 or 300 dots per inch). Libraries also exist for generating plot files from Fortran programs and for plotting on "dumb" devices such as a standard line printer.
<b>popen</b>	Dynamically allocates an array for file descriptors. The new signal interface is now used.
<b>psignal</b>	New signals have been added to the list.
<b>random</b>	An initialization bug that messed up default generation was fixed.
<b>rcmd</b>	Cleans up properly. A problem with doing multiple calls within one program was fixed.
<b>ruserok</b>	Now is more flexible about the format of <i>.rhosts</i> . Domain style hostnames do not need full specification if they are a part of the local domain, as determined by <i>hostname(1)</i> . <i>Ruserok</i> is more paranoid about ownership of <i>.rhosts</i> .
<b>scandir</b>	Handling of overflow has been fixed.
<b>setjmp</b>	The signal stack status is now set correctly.
<b>siginterrupt</b>	A new routine to set the signals for which system calls are not restarted after signal delivery.
<b>signal</b>	Keeps track of new features when changing signal handlers.
<b>sleep</b>	A couple of races have been fixed.
<b>stdio</b>	Has been modified to dynamically allocate slots for file pointers. Output on unbuffered files is now buffered within a call to <i>printf</i> or <i>fputs</i> for efficiency. <i>Fseek</i> now returns zero if it was successful. <i>Fread</i> and <i>fwrite</i> have been rewritten to improve performance. On the VAX, <i>fgets</i> , <i>gets</i> , <i>fputs</i> and <i>puts</i> were rewritten to take advantage of VAX string instructions and thus improve performance. Line buffering now works on any file descriptor, not just <i>stdout</i> and <i>stderr</i> . <i>Putc</i> is implemented completely within a macro except when the buffer is full or when a newline is output on a line-buffered file. Some sign extension bugs with the return value of <i>putc</i> have been fixed.
<b>string</b>	The routines <i>index</i> , <i>rindex</i> , <i>strcat</i> , <i>strcmp</i> , <i>strcpy</i> , <i>strlen</i> , <i>strncat</i> , and <i>strncpy</i> have been rewritten in VAX assembly language for efficiency. The C routines are included for use on other machines. Only <i>Makefiles</i> need to be modified to select the version to be used.
<b>syslog</b>	The third parameter to <i>openlog</i> is a "facility code" used to classify messages. References to <i>&lt;syslog.h&gt;</i> should be replaced with references to <i>&lt;sys/syslog.h&gt;</i> .
<b>ttyslot</b>	Uses the new <i>gettyent</i> routine.
<b>ualarm</b>	A simplified interface to <i>setitimer</i> , similar to <i>alarm</i> but with its argument in microseconds.
<b>usleep</b>	A new routine which resembles <i>sleep</i> but takes an argument in microseconds.



## Section 4

The system now supports the 64Kbit and 256Kbit RAM memory controllers for the VAX-11/780 and VAX-11/785, the second UNIBUS adapter for the VAX-11/750, and the new VAX 8600

with UNIBUS and/or MASSBUS peripherals. The Unibus management routines for network interfaces have been generalized in 4.3BSD; this change requires stylized changes within most of the network drivers. A number of changes were made to each terminal multiplexor driver as well. See sections 9 and 11 of the "Changes to the Kernel in 4.3BSD" document for details.

New manual entries in Section 4 have been created to describe the new communications protocols and network architectures that are supported. The most recent addition in 4.3BSD is the Xerox Network System protocols.

<b>arp</b>	<i>ioctl</i> s have been added to enter and delete entries in the Internet-to-Ethernet† address translation tables. Entries may be made permanent, and may be "published" to allow a host to act as an ARP server.
<b>ddn</b>	A new DDN Standard Mode X.25 IMP interface driver.
<b>de</b>	A new DEC DEUNA 10 Mb/s Ethernet interface driver.
<b>dhu</b>	A new DEC DHU-11 communications multiplexor driver.
<b>dmc</b>	The configuration flags may be used to specify how to set up the device. Multiple outstanding DMA requests can now be handled. A new encapsulation is used that allows multiple protocols to be supported, but is incompatible with that used by 4.2BSD and earlier Ultrix releases.
<b>dmz</b>	A new DEC DMZ-32 communications multiplexor driver.
<b>ec</b>	Has a corrected backoff algorithm. Multiple units are supported by placing the Unibus memory address in the device <i>flags</i> field.
<b>ex</b>	A new Excelan 204 10 Mb/s Ethernet interface driver.
<b>hdh</b>	A new ACC IF-11/HDH IMP interface driver.
<b>idp</b>	A description of the new Xerox Internet Datagram Protocol.
<b>il</b>	The driver has additional diagnostics and now supports Xerox NS.
<b>ip</b>	Support for IP options was added.
<b>ix</b>	A new Interlan NP100 10 Mb/s Ethernet interface driver.
<b>np</b>	A new device for downloading microcode into the Interlan NP100 10 Mb/s Ethernet interface driver.
<b>ns</b>	A description of the new Xerox Network Systems protocol family.
<b>nsip</b>	A description of the new software network interface encapsulating NS packets in IP packets.
<b>ps</b>	The driver for the Picture System 2 has a small change in interrupt handling.
<b>pty</b>	A new mode was added to allow a small set of commands to be passed to the pty master from the slave as a rudimentary type of <i>ioctl</i> , analogous to that of PKT mode. Using this mode or PKT mode, a <i>select</i> for exceptional conditions on the master side of a pty returns <b>true</b> when a command operation is available to be read. <i>Select</i> for writing on the master side has been fixed.
<b>spp</b>	A description of the new Xerox Sequenced Packet Protocol.
<b>tcp</b>	An option was added to disable small-packet avoidance under certain circumstances.
<b>tty</b>	PASS8 mode has been added to pass all 8 bits of input. New <i>ioctl</i> s were added to support the getting and setting of window size information for the terminal. A signal was added to notify processes when the window size changes.

† Ethernet is a trademark of Xerox Corporation.



## Section 5

A new subdirectory, */usr/include/protocols*, has been created to keep header files that are shared between user programs and daemons. Several header files have been moved here, including those for *rwhod*, *routed*, *timed*, *dump*, *talk*, and *restore*.

Two new header files, *<string.h>* and *<memory.h>*, have been added for System V compatibility.

<b>disktab</b>	Two new fields have been added to specify that the disk supports <i>bad144</i> -style bad sector forwarding, and that offsets should be specified by sectors rather than cylinders.
<b>dump</b>	The header file <i>&lt;dumprestor.h&gt;</i> has moved to <i>&lt;protocols/dumprestor.h&gt;</i> .
<b>gettytab</b>	New entries have been added, including a 2400 baud dial-in rotation for modems, a 19200 baud standard line, and an entry for the <i>xterm</i> terminal emulator of the <i>X</i> window system. New capabilities for automatic speed selection and setting strict xoff/xon flow control ( <i>decttlq</i> ) were added.
<b>termcap</b>	Many new entries were added and older entries fixed.
<b>ttys</b>	The format of the <i>ttys</i> file, <i>/etc/ttys</i> , reflects the merger of information previously kept in <i>/etc/ttys</i> , <i>/etc/securetty</i> , and <i>/etc/ttytype</i> . The new format permits arbitrary programs, not just <i>/etc/getty</i> , to be spawned by <i>init</i> . A special <i>window</i> field can be used to set up a window server before spawning a terminal emulator program.

## Section 6

<b>aardvark</b>	The "Dungeon Definition Language" processor has been updated to run on 4.3BSD, so that games such as <i>aardvark</i> now work again.
<b>battlestar</b>	A third generation adventure game.
<b>canfield</b>	The user interface has been improved so that one need not type so many carriage returns between games. Players are charged a maximum of three minutes of think time between moves should they put a game on hold for an extended period of time.
<b>fortune</b>	Has yet more adages (not better ones, just more).
<b>hunt</b>	The latest addition, a maze battle game for multiple players.
<b>mille</b>	Now plays slightly more intelligently, and prevents discarding of safeties.
<b>robots</b>	Much like the old game of chase, except different.
<b>rogue</b>	Has been made more of a scoundrel.

## Section 7

<b>hier</b>	Has been updated to reflect the reorganization to the user and system source.
<b>me</b>	Some new macros were added: <i>.sm</i> (smaller) and <i>.bu</i> (bulleted paragraph). The <i>pic</i> , <i>ideal</i> , and <i>gremlin</i> preprocessors are now supported.
<b>words</b>	Two new word lists have been added to <i>/usr/dict</i> . The 1935 Webster's word list is available as <i>web2</i> with a supplemental list in <i>web2a</i> .  Several hundred words have been added to <i>/usr/dict/words</i> , both general words ("abacus, capsize, goodbye, Hispanic, ...") and important technical terms (all the amino acids, many mathematical terms, a few dinosaurs, ...). About 10 spelling errors in <i>/usr/dict/words</i> have been corrected.  Several hundred words that <i>spell</i> derives without difficulty from existing words (e.g. "getting" from "get"), or that <i>spell</i> would accept anyway, e.g. "1st, 2nd" etc., have been removed from <i>/usr/dict/words</i> .



## Section 8

Major changes affecting system operations include:

- The format of the *ttys* file, */etc/ttys*, has been changed to include information about terminal type.
- The *crontab* file used by *cron* has a new field in each line to specify the user ID to be used.
- A new Internet server-server, *inetd*, listens for service requests on a number of ports and spawns the appropriate server upon demand. Fewer of the Internet services now require long-lived daemon processes.
- The *bad144* program can now be used to add new bad sectors to the bad sector file. Replacement sectors are rearranged as needed to sort the new sectors into the bad sector list. Reformat operations to mark bad sectors to the bad sector table should still be done only with the system running single user.
- *Getty's* description file, */etc/gettytab*, now describes what program should be run in addition to the other information that it used to include.

S  
M  
M  
12

arff	Has been extended to understand multiple directory segments. This allows it to handle the console RL02 pack on the VAX 8600.
arp	A new program for examining and modifying the kernel Address Resolution Protocol tables.
bad144	<i>Bad144</i> has new options to add sectors to the bad sector table and to attempt to copy sectors to their replacements before marking them bad. It verifies that the file is properly sorted. Verbose and no-write options allow dry runs.
catman	Now allows a list of manual directories. Links are properly set up so that the manual source need not be kept on line on all machines.
checkquota	Runs multiple filesystems in parallel. Quotas for users with zero blocks are left around but they are deleted if the user-id no longer exists.
chown	Was modified to be recursive. <i>Chown</i> accepts an <i>owner.group</i> syntax to change owner and group simultaneously. The group-id will be set correctly when dealing with symbolic links.
comsat	<i>Comsat</i> is now invoked by <i>inetd</i> . It reaps its child processes correctly. Large systems with many terminal lines are now handled.
config	Swap size may be specified. <i>Maxusers</i> is no longer truncated. The name of the generated <i>Makefile</i> is now capitalized. Object files may now be listed for inclusion in the <i>files</i> file and will be added to the compilation properly. Optional files may be listed multiple times if different options require their inclusion. <i>Swapconf</i> supports larger unit numbers. <i>Config</i> builds a new file containing definitions for counting device interrupts.
cron	<i>/usr/lib/crontab</i> has a new format to specify the user-id under which the process should be run.
diskpart	Handles disks with either cylinder or sector offsets and that do not use <i>bad144</i> bad block forwarding.
dump	When dumping at 6250 bpi, the tape is written in 32Kb records instead of 10Kb records. Efforts have been made to improve the consistency of dumps made on active file systems (though the practice is still NOT recommended). The Caltech streaming dump modifications using a ring of slave processes have been incorporated. <i>Dump</i> makes a better estimate of the size of the dump by attempting to account for files with holes. The error messages have been made less condescending.
edquota	Can edit quotas on filesystems where a user does not have any usage.

<b>fingerd</b>	A new daemon to return user information; it runs under <i>inetd</i> .
<b>fsck</b>	<i>Fsck</i> has been sped up considerably by eliminating one of the two passes across the inodes. It has also been taught to create and grow directories so that it can now rebuild the root of a file system as well as create and enlarge the <i>last+found</i> directory as necessary.
<b>ftpd</b>	Among the new facilities supported by the FTP server are: the ABOR command for transfer abort, the PASV command for third party transfers, and the new RFC959 FTP commands (such as STOU, "store unique"). <i>Ftpd</i> now uses <i>syslog</i> to log errors, and is invoked by <i>inetd</i> .
<b>gettable</b>	Now has a flag for checking the version without retrieving the whole host table.
<b>getty</b>	<i>Getty</i> supports automatic baud rate detection based on carriage return. Support for window system startup has been added. The login banner can now include the terminal name. The environment is set up now and passed to <i>login</i> .
<b>htable</b>	Some byte ordering problems have been fixed. It is more intelligent about gateway handling. A looping problem with single character host names has been fixed.
<b>ifconfig</b>	<i>Ifconfig</i> has been augmented to allow different address families. The current families understood are <i>inet</i> and <i>ns</i> . <i>Ifconfig</i> has additions to set up subnets of Internet networks, change Internet broadcast addresses, and set destination addresses of point-to-point links.
<b>implog</b>	Handles class B and class C networks.
<b>inetd</b>	A new program to spawn network servers on demand. <i>Inetd</i> listens on each port listed in its configuration file <i>/etc/inetd.conf</i> . When service requests arrive, it passes the original socket or a newly accepted socket to the designated server for the service. Several trivial services are implemented internally.
<b>init</b>	May run commands other than <i>getty</i> . Large systems are no longer a problem. Window systems may be started.
<b>lpc</b>	A new command, <i>down</i> , disables queueing and printing, and, optionally, creates a status message displayed by the <i>lpq</i> program. The <i>up</i> command reverses the effect of the <i>down</i> command. The <i>status</i> command now displays the contents of the print queue in addition to the status of the daemon process. The <i>clean</i> command does a better job of removing incomplete queue entries.
<b>lpd</b>	A new capability, <i>hl</i> , may be used to print a job's banner after the contents of the job. Error logging is now done with <i>syslog</i> (3). Hosts permitting remote access may now be specified in the file <i>/etc/hosts.lpd</i> (in addition to <i>/etc/hosts.equiv</i> ). A master lock file is now used so that <i>/dev/printer</i> can be automatically removed. Symbolic links to spool files are now checked carefully to close a security hole. All printing parameters are now properly reset for each job. Remote spooling connections now time out if the server crashes. Errors in spooling filters are now reported to users via mail. When servicing a remote job, files are not transferred unless enough disk space is available.
<b>mkfs</b>	Will print the filesystem information without creating the filesystem. Filesystem optimization may be specified.
<b>mkhosts</b>	A new program to rebuild the <i>/etc/hosts</i> dbm database. Note that this database is not used with the default name server configuration.
<b>mkpasswd</b>	A new program to rebuild the <i>/etc/passwd</i> dbm database.
<b>mount</b>	Better error messages are returned when <i>mount</i> fails. When checking <i>/etc/fstab</i> to find the device name of a file system when only the mount point is specified, it also checks the <i>type</i> field to insure that the entry is <i>rw</i> , <i>ro</i> , or <i>rq</i> .
<b>named</b>	Is a new program implementing the Internet domain naming system. It is used to perform hostname and address mapping functions for the standard C library functions, <i>gethostbyname</i> and <i>gethostbyaddr</i> if <i>named</i> is running.

<b>newfs</b>	Supports new options to <i>mkfs</i> .
<b>pac</b>	Has a new option, <i>-m</i> , to cause machine names to be disregarded in merging accounting information. The per-page cost is now taken from the printer description if it is not specified on the command line with the <i>-p</i> option.
<b>ping</b>	Is a new program for sending ICMP echo requests.
<b>pstat</b>	Can handle kernel crash dumps and new terminal multiplexers. Core dumps should be less frequent.
<b>repquota</b>	Only prints entries for users that have files (or blocks) allocated.
<b>restore</b>	The interactive mode of <i>restore</i> now understands globbing. Interrupting interactive mode returns to the prompt. A new input path name may be specified on each volume change. The tape block size is calculated dynamically unless it is specified with the <i>-b</i> flag on the command line.
<b>rexecd</b>	Now runs under <i>inetd</i> .
<b>rlogind</b>	Propagates window size changes in a backward compatible way. This is negotiated at startup time. <i>Inetd</i> now starts up the server.
<b>rmt</b>	Uses large network buffers for better performance.
<b>route</b>	Will handle subnets. Flags were added to specify whether a name is a host or a network. Multiple addresses are tried until an operation is successful or there are no more addresses to try.
<b>routed</b>	Is more strict about received packets' formats and values. Subnet routing is handled. Point to point links are handled. Gateways to external networks advertise a default route instead of all networks. The loopback network number is no longer compiled in. When a process is terminated, it tells its peers that its routes are no longer valid.
<b>rshd</b>	Is started by <i>inetd</i> . The address is passed through if the host name for the address cannot be determined.
<b>rwhod</b>	Should be less expensive to run. Broadcasts are done less frequently and path lookups are shorter. Large systems are handled better.
<b>rxformat</b>	Will now operate if the standard input is not a terminal.
<b>sa</b>	Supports alternate accounting files. The units of CPU time have changed.
<b>savecore</b>	Works correctly when given an alternate system name. Dump partitions smaller than the memory size are handled more gracefully.
<b>sendmail</b>	<p>Several bugs have been fixed. Upper case letters are allowed in file names and program arguments in the alias file. Multiple recipients sharing a receive program are not collapsed into one delivery. List owners on queued jobs have been fixed. Commas in quoted aliases work. Dollar signs in headers are no longer interpreted as macro expansions. Underscores are allowed in login names.</p> <p>Substantial performance enhancements have been made for large queues. If the <i>Y</i> option is not set, all jobs in the queue will be run in one process, with host statuses cached; this uses more memory but generally improves performance. The job priority now includes creation time and number of recipients (the <i>y</i> option) as well as the message size (the <i>q</i> option) and the job precedence (the <i>z</i> option); this priority is modified by the <i>Z</i> option whenever it fails to complete. No attempt is made to run large jobs if the load average is too high.</p> <p>The <i>\$[ ... \$]</i> syntax can be used on the RHS of a rewriting rule to canonicalize a host name using <i>gethostbyname</i>. This is especially useful when running the version of <i>gethostbyname</i> that calls the name server.</p> <p>Error reporting has been improved. Some limits have been increased. Security holes have been plugged. <i>Syslogd</i> and <i>vacation</i> are now part of the standard system.</p>

	Minor changes have been made to the configuration file. The RHS of aliases are no longer checked while the alias file is rebuilt unless the <code>n</code> option is set to improve performance. The character substituted for blanks in addresses is settable by the <code>B</code> option. The default network name (formerly hardwired "ARPA") is settable with the <code>N</code> option. The <code>E</code> mailer option escapes "From" lines with a <code>&gt;</code> on delivery (formerly the default to the local mailer).
<b>shutdown</b>	Has flags to specify that it should not sync the disks and that it should skip the disk checks after rebooting.
<b>swapon</b>	Error messages have been cleaned up and now specify the device to which they correspond.
<b>syslogd</b>	Formerly <i>syslog</i> , allows the classification of messages based on <i>facilities</i> . The configuration file has been restructured.
<b>talkd</b>	Now runs under <i>inetd</i> . New version, new protocol.
<b>telnetd</b>	Handles pty allocation better. <i>inetd</i> now starts the server. Interpretation of carriage return-newline now conforms with the standard, but is compatible with the 4.2BSD <i>telnet</i> client.
<b>tftpd</b>	Now works with other clients and is started by <i>inetd</i> .
<b>timed</b>	A new program for maintaining time synchronization between machines on a local network.
<b>trpt</b>	The <i>trpt</i> program to examine TCP traces now prints the traces in the correct order. It has been extended to follow traces as a connection runs.
<b>tunefs</b>	Supports the new filesystem optimization preferences.
<b>uucpd</b>	A new server, invoked by <i>inetd</i> , for running uucp over network connections.
<b>vipw</b>	Builds the new hashed lookup table. <i>/etc/passwd</i> will not be left unreadable if root has a restrictive umask.
<b>XNSrouted</b>	A new daemon, similar to <i>routed</i> , that implements the Xerox NS routing protocol.

## Appendix A – User Contributed Software

Several new programs have been contributed to the Berkeley distribution.

<b>ansitape</b>	Is a new program for handling tapes in ANSI format and for transferring files between UNIX and VMS.
<b>B</b>	Yet another new language.
<b>cpm</b>	Is a file transfer protocol between UNIX and CP/M.
<b>dipress</b>	A new program to convert <i>ditroff</i> output to Xerox Interpress format.
<b>emacs</b>	Is a public domain version of <i>emacs</i> .
<b>help</b>	An extensive new UNIX help facility.
<b>hyper</b>	A router and log program for the Hyperchannel.
<b>icon</b>	The latest and greatest version from Arizona.
<b>jove</b>	Is a simplified <i>emacs</i> -style editor.
<b>kermit</b>	A file transfer protocol between UNIX and microcomputers.
<b>mh</b>	This release includes MH Version 6.3, with Berkeley modifications. It has been rewritten numerous times since the original version release with 4.2BSD. Each utility is now infinitely programmable.
<b>mkmf</b>	Has been separated from <i>SPMS</i> .



<b>randf</b>	Is a new set of mail reading and transport programs.
<b>news</b>	The latest revision of the Usenet news programs, B news 2.10.3 beta.
<b>np100</b>	Utilities to download the Interlan NP100 Ethernet board.
<b>patch</b>	Is a new program designed for taking diffs and applying them to the source file. If you only look at one new program, this is the one!
<b>pathalias</b>	A new program that attempts to discover uucp path routing.
<b>pup</b>	An implementation of the Xerox PUP protocols and several useful programs that use them.
<b>rn</b>	A new interface for reading (or ignoring) news.
<b>sumacc</b>	A C compiler set of programs for doing MacIntosh software development.
<b>sunrpc</b>	Yet another RPC protocol.
<b>tac</b>	Is a program that displays a file in reverse line order.
<b>umodem</b>	Another file transfer protocol between UNIX and microcomputers.
<b>X</b>	A new window system that was developed at MIT. This distribution supports the DEC VS100, the Sun and the DEC b/w VAXStation II (QVSS).
<b>xns</b>	A courier RPC mechanism that runs on Xerox NS, and many useful applications developed at Cornell University.

## Changes to the Kernel in 4.3BSD

April 16, 1986

Michael J. Karels

Computer Systems Research Group  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California 94720

This document summarizes the changes to the kernel between the September 1983 4.2BSD distribution of UNIX† for the VAX‡ and the March 1986 4.3BSD release. It is intended to provide sufficient information that those who maintain the kernel, have local modifications to install, or who have versions of 4.2BSD modified to run on other hardware should be able to determine how to integrate this version of the system into their environment. As always, the source code is the final source of information, and this document is intended primarily to point out those areas that have changed.

Most of the changes between 4.2BSD and 4.3BSD fall into one of several categories. These are:

- bug fixes,
- performance improvements,
- completion of skeletal facilities,
- generalizations of the framework to accommodate new hardware and software systems, or to remove hardware- or protocol-specific code from common facilities, and
- new protocol and hardware support.

The major changes to the kernel are:

- the use of caching to decrease the overhead of filesystem name translation,
- a new interface to the *namei* name lookup function that encapsulates the arguments, return information and side effects of this call,
- removal of most of the Internet dependencies from common parts of the network, and greater allowance for the use of multiple address families on the same network hardware,
- support for the Xerox NS network protocols,
- support for the VAX 8600 and 8650 processors (with UNIBUS and MASSBUS peripherals, but not with CI bus or HSC50 disk controllers),
- new drivers for the DHU11 and DMZ32 terminal multiplexors, the TU81 and other TMSCP tape drives, the VS100 display, the DEUNA, Excelan 204, and Interlan NP100 Ethernet\* interfaces, and the ACC HDH and DDN X.25 IMP interfaces, and
- full support for the MS780-E memory controller on the VAX 11/780 and 11/785, using 64K and 256K memory chips.

This document is not intended to be an introduction to the kernel, but assumes familiarity with prior versions of the kernel. Other documents may be consulted for more complete discussions of the

† UNIX is a trademark of AT&T Bell Laboratories.

‡ DEC, VAX, PDP, MASSBUS, UNIBUS, Q-bus and ULTRIX are trademarks of Digital Equipment Corporation.

\* Ethernet is a trademark of Xerox Corporation.



kernel and its other subsystems. For more complete information on the internal structure and interfaces of the network subsystem, refer to "4.3BSD Networking Implementation Notes."

The author gratefully acknowledges the contributions of the other members of the Computer Systems Research Group at Berkeley and the other contributors to the work described here. Major contributors include Kirk McKusick, Sam Leffler, Jim Bloom, Keith Sklower, Robert Elz, and Jay Lepreau. Sam Leffler and Anne Hughes made numerous suggestions and corrections during the preparation of the manuscript.

## 1. General changes in the kernel

This section details some of the changes that affect multiple sections of the kernel.

### 1.1. Header files

The kernel is now compiled with an include path that specifies the standard location of the common header files, generally `/sys/h` or `./h`, and all kernel sources have had pathname prefixes removed from the `#include` directives for files in `./h` or the source directory. This makes it possible to substitute replacements for individual header files by placing them in the system compilation directory or in another directory in the include path.

### 1.2. Types

There have been relatively few changes in the types defined and used by the system. One significant exception is that new typedefs have been added for user ID's and group ID's in the kernel and common data structures. These typedefs, `uid_t` and `gid_t`, are both of type `u_short`. This change from the previous usage (explicit `short` ints) allows user and group ID's greater than 32767 to work reasonably.

### 1.3. Inline

The inline expansion of calls to various trivial or hardware-dependent operations has been a useful technique in the kernel. In prior releases this substitution was done by editing the assembly language output of the compiler with the sed script `asm.sed`. This technique has been refined in 4.3BSD by using a new program, `inline`, to perform the in-line code expansion and also optimize the code used to push the subroutine's operands; where possible, `inline` will merge stack pushes and pops into direct register loads. Also, this program performs the in-line code expansion significantly faster than the general-purpose stream editor it replaces.

### 1.4. Processor priorities

Functions to set the processor interrupt priority to block classes of interrupts have been used in UNIX on all processors, but the names of these routines have always been derived from the priority levels of the PDP11 and the UNIBUS. In order to clarify both the intent of elevated processor priority and the assumptions about their dependencies, all of the functions `splN`, where `N` is a small nonzero integer, have been renamed. In each case, the new name indicates the group of devices that are to be blocked from interrupts. The following table indicates the old and new names of these functions.

new name	devices blocked	old name	VAX IPL
<code>spl0</code>	none	<code>spl0</code>	0
<code>splsoftclock</code>	software clock interrupts	none	0x08
<code>splnet</code>	software network interrupts	<code>splnet</code>	0x0c
<code>spltty</code>	terminal multiplexors	<code>spl5</code>	0x15
<code>splbio</code>	disk and tape controllers	<code>spl5</code>	0x15
<code>splimp</code>	all network interfaces	<code>splimp</code>	0x16
<code>splclock</code>	interval timer	<code>spl6</code>	0x18
<code>splhigh</code>	all devices and state transitions	<code>spl7</code>	0x31



For use in device drivers only, UNIBUS priorities BR4 through BR7 may be set using the functions spl4, spl5, spl6 and spl7. Note that the latter two now correspond to VAX priorities 0x16 and 0x17 respectively, rather than the previous 0x18 and 0x1f priorities.

## 2. Header files

This section details changes in the header files in /sys/h.

acct.h	Process accounting is now done in units of 1/AHZ (64) seconds rather than seconds.
buf.h	The size of the buffer hash table has been increased substantially.
cmap.h	The core map has had a number of fields enlarged to support larger memories and filesystems. The limits imposed by this structure are now commented. The current limits are 64 Mb of physical memory, 255 filesystems, 1 Gb process segments, 8 Gb per filesystem, and 65535 processes and text entries. The machine-language support now derives its definitions of these limits and the cmap structure from this file.
dmap.h	The swap map per process segment was enlarged to allow images up to 64Mb.
domain.h	New entry points to each domain have been added, for initialization, externalization of access rights, and disposal of access rights.
errno.h	A definition of EDEADLK was added for System V compatibility.
fs.h	One spare field in the superblock was allocated to store an option for the fragment allocation policy.
inode.h	New fields were added to the in-core inode to hold a cache key and a pointer to any text image mapping the file. A new macro, ITIMES, is provided for updating the timestamps in an inode without writing the inode back to the disk. The inode is marked as modified with the IMOD flag. A flag has been added to allow serialization of directory renames.
ioctl.h	New <i>ioctl</i> operations have been added to get and set a terminal or window's size. The size is stored in a <i>winsize</i> structure defined here. Other new <i>ioctls</i> have been defined to pass a small set of special commands from pseudo-terminals to their controllers. A new terminal option, LPASS8, allows a full 8-bit data path on input. The two tablet line disciplines have been merged. A new line discipline is provided for use with IP over serial data lines.
mbuf.h	The handling of mbuf page clusters has been broken into macros separate from those that handle mbufs. MCLALLOC( <i>m</i> , <i>i</i> ) is used to allocate <i>i</i> mbuf clusters (where <i>i</i> is currently restricted to 1) and MCLFREE( <i>m</i> ) frees them. MCLGET( <i>m</i> ) adds a page cluster to the already-allocated mbuf <i>m</i> , setting the mbuf length to CLBYTES if successful. The new macro M_HASCL( <i>m</i> ) returns true if the mbuf <i>m</i> has an associated cluster, and MTOCL( <i>m</i> ) returns a pointer to such a cluster.
mtio.h	Definitions have been added for the TMSCP tape controllers and to enable or disable the use of an on-board tape buffer.
namei.h	This header file was renamed, completed and put into use.
param.h	Several limits have been increased. Old values are listed in parentheses after each item. The new limits are: 255 mounted filesystems (15), 40 processes per user (25), 64 open files (20), 20480 characters per argument list (10240), and 16 groups per user (8). The maximum length of a host name supported by the kernel is defined here as MAXHOSTNAMELEN. The default creation mask is now set to 022 by the kernel; previously that value was set by login, with the effect that remote shell processes used a different default. Clist blocks were doubled in size to 64 bytes.
proc.h	Pointers were added to the <i>proc</i> structure to allow process entries to be linked onto lists of active, zombie or free processes.
protosw.h	The address family field in the <i>protosw</i> structure was replaced with a pointer to the <i>domain</i> structure for the address family. Definitions were added for the arguments to



	the protocol <i>cloutput</i> routines.
signal.h	New signals have been defined for window size changes (SIGWINCH) and for user-defined functions (SIGUSR1 and SIGUSR2). The <i>sv_onstack</i> field in the <i>sigvec</i> structure has been redefined as a flags field, with flags defined for use of the signal stack and for signals to interrupt pending systems calls rather than restarting them. The <i>sigcontext</i> structure now includes the frame and argument pointers for the VAX so that the complete return sequence can be done by the kernel. A new macro, <i>sigmask</i> , is provided to simplify the use of <i>sigsetmask</i> , <i>sigblock</i> , and <i>sigpause</i> .
socket.h	Definitions were added for new options set with <i>setsockopt</i> . SO_BROADCAST requests permission to send to the broadcast address, formerly a privileged operation, while SO_SNDBUF and SO_RCVBUF may be used to examine or change the amount of buffer space allocated for a socket. Two new options are used only with <i>getsockopt</i> : SO_ERROR obtains any current error status and clears it, and SO_TYPE returns the type of the socket. A new structure was added for use with SO_LINGER. Several new address families were defined.
socketvar.h	The character and mbuf counts and limits in the <i>sockbuf</i> structure were changed from <i>short</i> to <i>u_short</i> . SB_MAX defines the limit to the amount that can be placed in a <i>sockbuf</i> . The <i>sosendallatonce</i> macro was corrected; it previously returned true for sockets using non-blocking I/O. <i>Soreadable</i> and <i>sowriteable</i> now return true if there is error status to report.
syslog.h	The system logging facility has been extended to allow kernel use, and the header file has thus been moved from /usr/include.
tablet.h	A new file that contains the definitions for use of the tablet line discipline.
text.h	Linkage fields have been added to the text structure for use in constructing a text table free list. The structure used in recording text table usage statistics is defined here.
time.h	The <i>time.h</i> header file has been split. Those definitions relating to the <i>gettimeofday</i> system call remain in this file, included as <i>&lt;sys/time.h&gt;</i> . The original <i>&lt;time.h&gt;</i> file has returned and contains the definitions for the C library time routines.
tty.h	The per-terminal data structure now contains the terminal size so that it can be changed dynamically. Files that include <i>&lt;sys/tty.h&gt;</i> now require <i>&lt;sys/ioctl.h&gt;</i> as well for the <i>winsize</i> structure definition.
types.h	The new typedefs for user and group ID's are located here. For compatibility and sensibility, the <i>size_t</i> , <i>time_t</i> and <i>off_t</i> types have all been changed from <i>int</i> to <i>long</i> . New definitions have been added for integer masks and bit operators for use with the <i>select</i> system call.
uio.h	The offset field in the <i>uio</i> structure was changed from <i>int</i> to <i>off_t</i> . Manifest constants for the <i>uio</i> segment values are now provided.
un.h	The path in the Unix-domain version of a <i>sockaddr</i> was reduced so that use of the entire pathname array would still allow space for a null after the structure when stored in an mbuf.
unpcb.h	A Unix-domain socket's own address is now stored in the protocol control block rather than that of the socket to which it is connected. Fields have been added for flow control on stream connections. If a <i>stat</i> has caused the assignment of a dummy inode number to the socket, that number is stored here.
user.h	The user ID's, group ID's and groups array are declared using the new types for these ID's. A new field was added to handle the new signal flag avoiding system call restarts. The index of the last used file descriptor for the process is maintained in <i>u.u_lastfile</i> . The global fields <i>u_base</i> , <i>u_count</i> , and <i>u_offset</i> have been eliminated, with the new <i>nameidata</i> structure replacing their remaining function. The <i>a.out</i> header is no longer kept in the user structure.

<b>vmmac.h</b>	Several macros have been rewritten to improve the code generated by the compiler. New macros were added to lock and unlock <i>cmap</i> entries, substituting for <i>mlock</i> and <i>munlock</i> .
<b>vmmeter.h</b>	All counters are now uniformly declared as <i>long</i> . Software interrupts are now counted.

### 3. Changes in the kernel proper

The next several sections describe changes in the parts of the kernel that reside in */sys/sys*. This section summarizes several of the changes that impact several different areas.

#### 3.1. Process table management

Although the process table has grown considerably since its original design, its use was largely the same as in its first incarnation. Several parts of the system used a linear search of the entire table to locate a process, a group of processes, or group of processes in a certain state. 4.2BSD maintained linkages between the children of each parent process, but made no use of these pointers. In order to reduce the time spent examining the process table, several changes have been made. The first is to place all process table entries onto one of three doubly-linked lists, one each for entries in use by existing processes (*allproc*), entries for zombie processes (*zombproc*), and free entries (*freeproc*). This allows the scheduler and other facilities that must examine all existing processes to limit their search to those entries actually in use. Other searches are avoided by using the linkage among the children of each process and by noting a range of usable process ID's when searching for a new unique ID.

#### 3.2. Signals

One of the major incompatibilities introduced in 4.2BSD was that system calls interrupted by a caught signal were restarted. This facility, while necessary for many programs that use signals to drive background activities without disrupting the foreground processing, caused problems for other, more naive, programs. In order to resolve this difficulty, the 4.2BSD signal model has been extended to allow signal handlers to specify whether or not the signal is to abort or to resume interrupted system calls. This option is specified with the *sigvec* call used to specify the handler. The *sv\_onstack* field has been usurped for a flag field, with flags available to indicate whether the handler should be invoked on the signal stack and whether it should interrupt pending system calls on its return. As a result of this change, those system calls that may be restarted and that therefore take control over system call interruptions must be modified to support this new behavior. The calls affected in 4.3BSD are *open*, *read/write*, *ioctl*, *flock* and *wait*.

Another change in signal usage in 4.3BSD affects fewer programs and less kernel code. In 4.2BSD, invocation of a signal handler on the signal stack caused some of the saved status to be pushed onto the normal stack before switching to the signal stack to build the call frame. The status information on the normal stack included the saved PC and PSL; this allowed a user-mode *rei* instruction to be used in implementing the return to the interrupted context. In order to avoid changes to the normal runtime stack when switching to the signal stack, the return procedure has been changed. As the return mechanism requires a special system call for restoring the signal state, that system call was replaced with a new call, *sigreturn*, that implements the complete return to the previous context. The old call, number 139, remains in 4.3BSD for binary compatibility with the 4.2BSD version of *longjmp*.

#### 3.3. Open file handling

Previous versions of UNIX have traditionally limited each process to at most 20 files open simultaneously. In 4.2BSD, that limit could not be increased past 30, as a 5-bit field in the page table entry was used to specify either a file number or the reserved values PGTEXT or PGZERO (fill from text file or zero fill). However, the file mapping facility that previously used this field no longer existed, and its replacement is unlikely to require this low limit. Accordingly, the internal virtual memory system support for mapped files has been removed and the number of open files increased. The standard limit is 64, but this may easily be increased if sufficient memory for the user structure is



provided. In order to avoid searching through this longer list of open files when the actual number in use is small, the index of the last used open file slot is maintained in the field *u.u\_lastfile*. The routines that implement open and close or implicit close (*exit* and *exec*) maintain this field, and it is used whenever the open file array *u.u\_ofile* is scanned.

### 3.4. Niceness

The values for *nice* used in 4.2BSD and previous systems ranged from 0 though 39. Each use of this scheduling parameter offset the actual value by the default, NZERO (20). This has been changed in 4.3BSD to use a range of -20 to 20, with NZERO redefined as zero.

### 3.5. Software interrupts and terminal multiplexors

The DH11 and DZ11 terminal multiplexor handlers had been modified to use the hardware's received-character silo when those devices were used by the Berknet network. In order to avoid stagnation of input characters and slow response to input during periods of reduced input, the low-level software clock interrupt handler had been made to call the terminal drivers to drain input. When the clock rate was increased in 4.2BSD, the overhead of checking the input silos with each clock tick was increased, and the use of specialized network hardware reduced the need for this optimization. Therefore, the terminal multiplexors in 4.3BSD use per-character interrupts during periods of low input rate, and enable the silos only during periods of high-speed input. While the silo is enabled, the routine to drain it runs less frequently than every clock tick; it is scheduled using the standard timeout mechanism. As a result, the software clock service routine need not to be invoked on every clock tick, but only when timeouts or profiling require service.

### 3.6. Changes in initialization and kernel-level support

This section describes changes in the kernel files in */sys/sys* with prefixes *init\_* or *kern\_*.

init_main.c	Several subsystems have new or renamed initialization routines that are called by <i>main</i> . These include <i>pqinit</i> for process queues, <i>xinit</i> for the text table handling routines, and <i>nchinit</i> for the name translation cache. The virtual memory startup <i>setupclock</i> has been replaced by <i>vminit</i> , that also sets the initial virtual memory limits for process 0 and its descendants. Process 1, <i>init</i> , is now created before process 2, <i>pagedaemon</i> .
init_sysent.c	In addition to entries for the two system calls new in 4.3BSD, the system call table specifies a range of system call numbers that are reserved for redistributors of 4.3BSD. Other unused slots in earlier parts of the table should be reserved for future Berkeley use. Syscall 63 is no longer special.
kern_acct.c	The process time accounting file in 4.2BSD stored times in seconds rather than clock ticks. This made accounting independent of the clock rate, but was too large a granularity to be useful. Therefore, 4.3BSD uses a smaller but unvarying unit for accounting times, 1/64 second, specified in <i>acct.h</i> as its reciprocal AHZ. The <i>compress</i> function converts seconds and microseconds to these new units, expressed as before in 16-bit pseudo-floating point numbers.
kern_clock.c	The hardware clock handler implements the new time-correction primitive <i>adjtime</i> by skewing the rate at which time increases until a specified correction has been achieved. The <i>bump</i> routine used to increment the time has been changed into a macro. The overhead of software interrupts used to schedule the <i>softclock</i> handler has been reduced by noting whether any profiling or timeout activity requires it to run, and by calling <i>softclock</i> directly from <i>hardclock</i> (with reduced processor priority) if the previous priority was sufficiently low.
kern_descrip.c	Most uses of the <i>getf()</i> function have been replaced by the GETF macro form. The <i>dup</i> calls (including that from <i>fcntl</i> ) no longer copy the close-on-exec flag from the original file descriptor. Most of the changes to support the open file descriptor high-water mark, <i>u.u_lastfile</i> , are in this file. The <i>flock</i> system call has had several bugs

- fixed. Unix-domain file descriptor garbage collection is no longer triggered from *closef*, but when a socket is torn down.
- kern\_exec.c** The *a.out* header used in the course of *exec* is no longer in the user structure, but is local to *exec*. Argument and environment strings are copied to and from the user address space a string at a time using the new *copyinstr* and *copyoutstr* primitives. When invoking an executable script, the first argument is now the name of the interpreter rather than the file name; the file name appears only after the interpreter name and optional argument. An *iput* was moved to avoid a deadlock when the executable image had been opened and marked close-on-exec. The *setregs* routine has been split; machine-independent parts such as signal action modification are done in *execve* directly, and the remaining machine-dependent routine was moved to *machdep.c*. Image size verification using *chksize* checks data and bss sizes separately to avoid overflow on their addition.
- kern\_exit.c** Instead of looping at location 0x13 in user mode if */etc/init* cannot be executed, the system now prints a message and pauses. This is done by *exit* if process 1 could not run. The search for child processes in *exit* uses the child and sibling linkage in the *proc* entry instead of a linear search of the *proc* table. Failures when copying out resource usage information from *wait* are now reflected to the caller.
- kern\_fork.c** One of the two linear searches of the *proc* table during process creation has been eliminated, the other looks only at active processes. As the first scan is needed only to count the number of processes for this user, it is bypassed for root. A comment dating to version 7 ("Partially simulate the environment so that when it is actually created (by copying) it will look right.") has finally been removed; it relates only to PDP-11 code.
- kern\_mman.c** *Chksize* takes an extra argument so that data and bss expansion can be checked separately to avoid problems with overflow.
- kern\_proc.c** The *spgrp* routine has been corrected. An attempt to optimize its  $O(n^2)$  algorithm (multiple scans of the process table) did so incorrectly; it now uses the child and sibling pointers in the *proc* table to find all descendents in linear time. *Pqinit* is called at initialization time to set up the process queues and free all process slots.
- kern\_prot.c** A number of changes were needed to reflect the type changes of the user and group ID's. The *getgroups* and *setgroups* routines pass groups as arrays of integers and thus must convert. All scans of the groups array look for an explicit NOGROUP terminator rather than any negative group. For consistency, the *setreuid* call sets the process *p\_uid* to the new effective user ID instead of the real ID as before. This prevents the anomaly of a process not being allowed to send signals to itself.
- kern\_resource.c** Attempts to change resource limits for process sizes are checked against the maximum segment size that the swap map supports, *maxdmap*. The error returned when attempting to change another user's priority was changed from EACCESS to EPERM.
- kern\_sig.c** The *sigmask* macro is now used throughout the kernel. The treatment of the *sigvec* flag has been expanded to include the SV\_INTERRUPT option. *Kill* and *killpg* have been rewritten, and the errors returned are now closer to those of System V. In particular, unprivileged users may broadcast signals with no error if they managed to kill something, and an attempt to signal process group 0 (one's own group) when no group is set receives an ESRCH instead of an EINVAL. SIGWINCH joins the class of signals whose default action is to ignore. When a process stops under *ptrace*, its parent now receives a SIGCHLD.
- kern\_synch.c** The CPU overhead of *schedcpu* has been reduced as much as possible by removing loop invariants and by ignoring processes that have not run since the last calculation. When long-sleeping processes are awakened, their priority is recomputed to consider their sleep time. *Schedcpu* need not remove processes with new priorities from their run queues and reinsert them unless they are moving to a new queue. The sleep



	queues are now treated as circular (FIFO) lists, as the old LIFO behavior caused problems for some programs queued for locks. <i>Sleep</i> no longer allows context switches after a panic, but simply drops the processor priority momentarily then returns; this converts sleeps during the filesystem update into busy-waits.
kern_time.c	<i>Gettimeofday</i> returns the microsecond time on hardware supporting it, including the VAX. It is now possible to set the timezone as well as the time with <i>settimeofday</i> . A system call, <i>adjtime</i> , has been added to correct the time by a small amount using gradual skew rather than discontinuous jumps forward or backward.
kern_xxx.c	The 4.1-compatible <i>signal</i> entry sets the signal SV_INTERRUPT option as well as the per-process SOUSIG, which now controls only the resetting of signal action to default upon invocation of a caught signal.
subr_log.c	This new file contains routines that implement a kernel error log device. Kernel messages are placed in the message buffer as before, and can be read from there through the log device <i>/dev/klog</i> .
subr_mcount.c	The kernel profiling buffers are allocated with <i>calloc</i> instead of <i>wmemall</i> to avoid the dramatic decrease in user virtual memory that could be supported after allocation of a large section of <i>usrpt</i> .
subr_prf.c	Support was added for the kernel error log. The <i>log</i> routine is similar to <i>printf</i> but does not print on the console, thereby suspending system operation. <i>Log</i> takes a priority as well as a format, both of which are read from the log device by the system error logger <i>syslogd</i> . <i>Uprintf</i> was modified to check its terminal output queue and to block rather than to use all of the system clists; it is now even less appropriate for use from interrupt level. <i>Tprintf</i> is similar to <i>uprintf</i> but prints to the tty specified as an argument rather than to that of the current user. <i>Tprintf</i> does not block if the output queue is overfull, but logs only to the error log; it may thus be used from interrupt level. Because of these changes, <i>putchar</i> and <i>printr</i> require an additional argument specifying the destination(s) of the character. The <i>tablefull</i> error routine was changed to use <i>log</i> rather than <i>printf</i> .
subr_rmap.c	An off-by-one error in <i>rmget</i> was corrected.
sys_generic.c	The <i>select</i> call may now be used with more than 32 file descriptors, requiring that the masks be treated as arrays. The result masks are returned to the user if and only if no error (including EINTR) occurs. A select bug that caused processes to disappear was fixed; <i>selwakeup</i> needed to handle stopped processes differently than sleeping processes.
sys_inode.c	Problems occurring after an interrupted close were corrected by forcing <i>ino_close</i> to return to <i>closef</i> even after an interrupt; otherwise, <i>f_count</i> could be cleared too early or twice. The code to unhash text pages being overwritten needed to be protected from memory allocations at interrupt level to avoid a bogus "panic: munhash." The internal routine implementing <i>flock</i> was reworked to avoid several bad assumptions and to allow restarts after an interruption.
sys_process.c	<i>Procxmt</i> uses the new <i>ptrace.h</i> header file; hopefully, the next release will have neither <i>ptrace</i> nor <i>procxmt</i> . The text XTRC flag is set when modifying a pure text image, protecting it from sharing and overwriting.
sys_socket.c	The socket involved in an interface <i>ioctl</i> is passed to <i>ifioctl</i> so that it can call the protocol if necessary, as when setting the interface address for the protocol. It is now possible to be notified of pending out-of-band data by selecting for exceptional conditions.
syscalls.c	The system call names here have been made to agree with reality.

## 3.7. Changes in the terminal line disciplines

- tty.c** The kernel maintains the terminal or window size in the `tty` structure and provides *ioctl*s to set and get these values. The window size is cleared on final close. The sizes include rows and columns in characters and may include X and Y dimensions in pixels where that is meaningful. The kernel makes no use of these values, but they are stored here to provide a consistent way to determine the current size. When a new value is set, a SIGWINCH signal is sent to the process group associated with the terminal.
- The notions of line discipline exit and final close have been separated. *Ttyclose* is used only at final close, while *ttyclose* is provided for closing down a discipline. Modem control transitions are handled more cleanly by moving the common code from the terminal hardware drivers into the line disciplines; the *l\_modem* entry in the *linesw* is now used for this purpose. *Ttymodem* handles carrier transitions for the standard disciplines; *nullmodem* is provided for disciplines with minimal requirements.
- A new mode, LPASS8, was added to support 8-bit input in normal modes; it is the input analog of LLITOUT. An entry point, *checkoutq*, has been added to enable internal output operations (*uprintf*, *tprintf*) to check for output overflow and optionally to block to wait for space. Certain operations are handled more carefully than before: the use of the TIOCSTI *ioctl* requires read permission on the terminal, and SPGRP is disallowed if the group corresponds with another user's process. *Ttread* and *ttwrite* both check for carrier drop when restarting after a sleep. An off-by-one consistency check of *uio\_iovcnt* in *ttwrite* was corrected. A bug was fixed that caused data to be flushed when opening a terminal that was already open when using the "old" line discipline. *Select* now returns true for reading if carrier has been lost. While changing line disciplines, interrupts must be disabled until the change is complete or is backed out. If changing to the same discipline, the close and reopen (and probable data flush) are avoided. The *t\_delct* field in the `tty` structure was not used and has been deleted.
- tty\_conf.c** The line discipline close entries that used *ttyclose* now use *ttyclose*. The two tablet disciplines have been combined. A new entry was added for a Serial-Line link-layer encapsulation for the Internet Protocol, SLIPDISC.
- tty\_pty.c** Large sections of the pseudo-tty driver have been reworked to improve performance and to avoid races when one side closed, which subsequently hung pseudo-terminals. The line-discipline modem control routine is called to clean up when the master closes. Problems with REMOTE mode and non-blocking I/O were fixed by using the raw queue rather than the canonicalized queue. A new mode was added to allow a small set of commands to be passed to the pty master from the slave as a rudimentary type of *ioctl*, in a manner analogous to that of PKT mode. Using this mode or PKT mode, a *select* for exceptional conditions on the master side of a pty returns true when a command operation is available to be read. *Select* for writing on the master side has been corrected, and now uses the same criteria as *ptcwrite*. As the pty driver depends on normal operation of the tty queues, it no longer permits changes to non-tty line disciplines.
- tty\_subr.c** The *clist* support routines have been modified to use block moves instead of *getc/putc* wherever possible.
- tty\_tablet.c** The two line disciplines have been merged and a number of new tablet types are supported. Tablet type and operating mode are now set by *ioctl*s. Tablets that continuously stream data are now told to stop sending on last close.

#### 4. Changes in the filesystem

The major change in the filesystem was the addition of a name translation cache. A table of recent name-to-inode translations is maintained by *namei*, and used as a lookaside cache when translating each component of each file pathname. Each *namecache* entry contains the parent directory's device and inode, the length of the name, and the name itself, and is hashed on the name. It also contains a pointer to the inode for the file whose name it contains. Unlike most inode pointers, which hold a "hard" reference by incrementing the reference count, the name cache holds a "soft" reference, a pointer to an inode that may be reused. In order to validate the inode from a name cache reference, each inode is assigned a unique "capability" when it is brought into memory. When the inode entry is reused for another file, or when the name of the file is changed, this capability is changed. This allows the inode cache to be handled normally, releasing inodes at the head of the LRU list without regard for name cache references, and allows multiple names for the same inode to be in the cache simultaneously without complicating the invalidation procedure. An additional feature of this scheme is that when opening a file, it is possible to determine whether the file was previously open. This is useful when beginning execution of a file, to check whether the file might be open for writing, and for similar situations.

Other changes that are visible throughout the filesystem include greater use of the *ILOCK* and *IUNLOCK* macros rather than the subroutine equivalents. The inode times are updated on each *irele*, not only when the reference count reaches zero, if the *IACC*, *IUPD* or *ICHG* flags are set. This is accomplished with the *ITIMES* macro; the inode is marked as modified with the new *IMOD* flag, that causes it to be written to disk when released, or on the next sync.

The remainder of this section describes the filesystem changes that are localized to individual files.

##### *ufs\_alloc.c*

The algorithm for extending file fragments was changed to take advantage of the observation that fragments that were once extended were frequently extended again, that is, that the file was being written in fragments. Therefore, the first time a given fragment is allocated, a best-fit strategy is used. Thereafter, when this fragment is to be extended, a full-sized block is allocated, the fragment removed from it, and the remainder freed for use in subsequent expansion. As this policy may result in increased fragmentation, it is not used when the filesystem becomes excessively fragmented (i.e. when the number of free fragments falls to 2% of the minfree value); the policy is stored in the superblock and may be changed with *tunefs*. The *fserr* routine was converted to use *log* rather than *printf*.

##### *ufs\_bio.c*

I/O operations traced now include the size where relevant.

##### *ufs\_inode.c*

The size of the buffer hash table was increased substantially and changed to a power of two to allow the modulus to be computed with a mask operation. *Iget* invalidates the capability in each inode that is flushed from the inode cache for reuse. The new *igrab* routine is used instead of *iget* when fetching an inode from a name cache reference; it waits for the inode to be unlocked if necessary, and removes it from the free list if it was free. The caller must check that the inode is still valid after the *igrab*. A bug was fixed in *itrunc* that allowed old contents to creep back into a file. When truncating to a location within a block, *itrunc* must clear the remainder of the block. Otherwise, if the file is extended by seeking past the end of file and then writing, the old contents reappear.

##### *ufs\_mount.c*

The *mount* system call was modified to return different error numbers for different types of errors. *Mount* now examines the superblock more carefully before using size field it contains as the amount to copy into a new buffer. If a mount fails for a reason other than the device already being mounted, the device is closed again. When performing the name lookup for the mount point, *mount* must prevent the name translation from being left in the name cache; *umount* must flush all name translations for the device. A bug in *getmdev* caused an inode to remain locked if the specified device was not a block special file; this has been fixed.



- ufs\_namei.c** This file was previously called `ufs_nami.c`. The *namei* function has a new calling convention with its arguments, associated context, and side effects encapsulated in a single structure. It has been extensively modified to implement the name cache and to cache directory offsets for each process. It may now return ENAMETOOLONG when appropriate, and returns EINVAL if the 8th bit is set on one of the pathname characters. Directories may be foreshortened if the last one or more blocks contain no entries; this is done when files are being created, as the entire directory must already be searched. An entry is provided for invalidating the entire name cache when the 32-bit prototype for capabilities wraps around. This is expected to happen after 13 months of operation, assuming 100 name lookups per second, all of which miss the cache.
- A change in filesystem semantics is the introduction of “sticky” directories. If the ISVTX (sticky text) bit is set in the mode of a directory, files may only be removed from that directory by the owner of the file, the owner of the directory, or the superuser. This is enforced by *namei* when the lookup operation is DELETE.
- ufs\_subr.c** The strategy for *syncip*, the internal routine implementing *fsync*, has been modified for large files (those larger than half of the buffer cache). For large files all modified buffers for the device are written out. The old algorithm could run for a very long time on a very large file, that might not actually have many data blocks. The *update* routine now saves some work by calling *iupdate* only for modified inodes. The C replacements for the special VAX instructions have been collected in this file.
- ufs\_syscalls.c** When doing an open with flags O\_CREAT and O\_EXCL (create only if the file did not exist), it is now considered to be an error if the target exists and is a symbolic link, even if the symbolic link refers to a nonexistent file. This behavior is desirable for reasons of security in programs that create files with predictable names. *Rename* follows the policy of *namei* in disallowing removal of the target of a rename if the target directory is “sticky” and the user is not the owner of the target or the target directory. A serious bug in the open code which allowed directories and other unwritable files to be truncated has been corrected. Interrupted opens no longer lose file descriptors. The *lseek* call returns an EPIPE error when seeking on sockets (including pipes) for backward compatibility. The error returned from *readlink* when reading something other than a symbolic link was changed from ENXIO to EINVAL. Several calls that previously failed silently on read-only filesystems (*chmod*, *chown*, *fchmod*, *fchown* and *utimes*) now return EROFS. The *rename* code was reworked to avoid several races and to invalidate the name cache. It marks a directory being renamed with IRENAME to avoid races due to concurrent renames of the same directory. *Mkdir* now sets the size of all new directories to DIRBLKSIZE. *Rmdir* purges the name cache of entries for the removed directory.
- ufs\_XXX.c** The routines *uchar* and *schar* are no longer used and have been removed.
- quota\_kern.c** The quota hash size was changed to a power of 2 so that the modulus could be computed with a mask.
- quota\_ufs.c** If a user has run out of warnings and had the hard limit enforced while logged in, but has then brought his allocation below the hard limit, the quota system reverts to enforcing the soft limit, and resets the warning count; users previously were required to log out and in again to get this affect.

#### 4.1. Changes in Interprocess Communication support

- uipc\_domain.c** The skeletal support for the PUP-1 protocol has been removed. A domain for Xerox NS is now in use. The per-domain data structure allows a per-domain initialization routine to be called at boot time.
- The *pfindproto* routine, used in creating a socket to support a specified protocol, takes an additional argument, the type of the socket. It checks both the protocol and



type, useful when the same protocol implements multiple socket types. If the type is `SOCK_RAW` and no exact match is found, a *protosw* entry for raw support and a wildcard protocol (number zero) will be used. This allows for a generic raw socket that passes through packets for any given protocol.

The second argument to *pfctlinput*, the generic error-reporting routine, is now declared as a *sockaddr* pointer.

#### uipc\_mbuf.c

The mbuf support routines now use the *wait* flag passed to *m\_get* or `MGET`. If `M_WAIT` is specified, the allocator may wait for free memory, and the allocation is guaranteed to return an mbuf if it returns. In order to prevent the system from slowly going to sleep after exhausting the mbuf pool by losing the mbufs to a leak, the allocator will panic after creating the maximum allocation of mbufs (by default, 256K). Redundant *spl*'s have been removed; most internal routines must be called at *splimp*, the highest priority at which mbuf and memory allocation occur.

When copying mbuf chains *m\_copy* now preserves the type of each mbuf. There were problems in *m\_adj*, in particular assumptions that there would be no zero-length mbufs within the chain; this was corrected by changing its *n*-pass algorithm for trimming from the tail of the chain to either one- or two-pass, depending on whether the correction was entirely within the last mbuf. In order to avoid return business, *m\_pullup* was changed to pull additional data (`MPULL_EXTRA`, defined in *mbuf.h*) into the contiguous area in the first mbuf, if convenient. *m\_pullup* will use the first mbuf of the chain rather than a new one if it can avoid copying.

#### uipc\_pipe.c

This "temporary" file has been removed; pipe now uses *socketpair*.

#### uipc\_proto.c

New entries in the protocol switch for externalization and disposal of access rights are initialized for the Unix domain protocols.

#### uipc\_socket.c

The *socreate* function uses the new interface to *pfindproto* described above if the protocol is specified by the caller. The *soconnect* routine will now try to disconnect a connected socket before reconnecting. This is only allowed if the protocol itself is not connection oriented. Datagram sockets may connect to specify a default destination, then later connect to another destination or to a null destination to disconnect. The *sodisconnect* routine never used its second argument, and it has been removed.

The *sosend* routine, which implements write and send on sockets, has been restructured for clarity. The old routine had the main loop upside down, first emptying and then filling the buffers. The new implementation also makes it possible to send zero-length datagrams. The maximum length calculation was simplified to avoid problems trying to account for both mbufs and characters of buffer space used. Because of the large improvement in speed of data handling when large buffers are used, *sosend* will use page clusters if it can use at least half of the cluster. Also, if not using nonblocking I/O, it will wait for output to drain if it has enough data to fill an mbuf cluster but not enough space in the output queue for one, instead of fragmenting the write into small mbufs. A bug allowing access rights to be sent more than once when using scatter-gather I/O (*sendmsg*) was fixed. A race that occurred when *uiomove* blocked during a page fault was corrected by allowing the protocol send routines to report disconnection errors; as with disconnection detected earlier, *sosend* returns `EPIPE` and sends a `SIGPIPE` signal to the process.

The receive side of socket operations, *soreceive*, has also been reworked. The major changes are a reflection of the way that datagrams are now queued; see *uipc\_socket2.c* for further information. The `MSG_PEEK` flag is passed to the protocol's *usrreq* routine when requesting out-of-band data so that the protocol may know when the out-of-band data has been consumed. Another bug in access-rights passing was corrected here; the protocol is not called to externalize the data when `PEEKing`.

The *soseopt* and *sogetopt* functions have been expanded considerably. The options that existed in 4.2BSD all set some flag at the socket level. The corresponding options in 4.3BSD use the value argument as a boolean, turning the flag off or on as appropriate. There are a number of additional options at the socket level. Most importantly, it is possible to adjust the send or receive buffer allocation so that higher throughput may be achieved, or that temporary peaks in datagram arrival are less likely to result in datagram loss. The linger option is now set with a structure including a boolean (whether or not to linger) and a time to linger if the boolean is true. Other options have been added to determine the type of a socket (eg, *SOCK\_STREAM*, *SOCK\_DGRAM*), and to collect any outstanding error status. If an option is not destined for the socket level itself, the option is passed to the protocol using the *ctloutput* entry. *Getopt*'s last argument was changed from *mbuf \** to *mbuf \*\** for consistency with *setopt* and the new *ctloutput* calling convention.

*Select* for exceptional conditions on sockets is now possible, and this returns true when out-of-band data is pending. This is true from the time that the socket layer is notified that the OOB data is on its way until the OOB data has been consumed. The interpretation of socket process groups in 4.2BSD was inconsistent with that of ttys and with the *fcntl* documentation. This was corrected; positive numbers refer to processes, negative numbers to process groups. The socket process group is used when posting a SIGURG to notify processes of pending out-of-band data.

**uipc\_socket2.c** Signal-driven I/O now works with sockets as well as with ttys; *sorwakeup* and *sowakeup* call the new routine *sowakeup* which calls *sbwakeup* as before and also sends SIGIO as appropriate. Process groups are interpreted in the same manner as for SIGURG.

Larger socket buffers may be used with 4.3BSD than with 4.2BSD; socket buffers (*sockbufs*) have been modified to use unsigned short rather than short integers for character counts and mbuf counts. This increases the maximum buffer size to 64K-1. These fields should really be unsigned longs, but a socket would no longer fit in an mbuf. So that as much as possible of the allotment may be used, *sbreserve* allows the high-water mark for data to be set as high as 80% of the maximum value (64K), and sets the high-water mark on mbuf allocation to the smaller of twice the character limit and 64K.

In 4.2BSD, datagrams queued in sockbufs were linked through the mbuf *m\_next* field, with *m\_act* set to 1 in the last mbuf of each datagram. Also, each datagram was required to have one mbuf to contain an address, another to contain access rights, and at least one additional mbuf of data. In 4.3BSD, the mbufs comprising a datagram are linked through *m\_next*, and different datagrams are linked through the *m\_act* field of the first mbuf in each. No mbuf is used to represent missing components of a datagram, but the ordering of the mbufs remains important. The components are distinguished by the mbuf type. Any address must be in the first mbuf. Access rights follow the address if present, otherwise they may be first. Data mbufs follow; at least one data buffer will be present if there is no address or access rights. The routines *sbappend*, *sbappendaddr*, *sbappendrights* and *sbappendrecord* are used to add new data to a sockbuf. The first of these appends to an existing record, and is commonly used for stream sockets. The other three begin new records with address, optional rights, and data (*sbappendaddr*), with rights and data (*sbappendrights*), or data only (*sbappendrecord*). A new internal routine, *sbcompress*, is used by these functions to compress and append data mbufs to a record. These changes improve the functionality of this layer and in addition make it faster to find the end of a queue.

An occasional "panic: sbdrop" was due to zero-length mbufs at the end of a chain. Although these should no longer be found in a sockbuf queue, *sbdrop* was fixed to free empty buffers at the end of the last record. Similarly, *sbfree* continues to empty



a sockbuf as long as mbufs remain, as zero-length packets might be present. *Sbdroprecord* was added to free exactly one record from the front of a sockbuf queue.

**uipc\_syscalls.c** Errors reported during an *accept* call are cleared so that subsequent *accept* calls may succeed. A failed attempt to *connect* returns the error once only, and SOISCONNECTING is cleared, so that additional connect calls may be attempted. (Lower level protocols may or may not allow this, depending on the nature of the failure.) The *socketpair* system call has been fixed to work with datagram sockets as well as with streams, and to clean up properly upon failure. Pipes are now created using *connect2*. An additional argument, the type of the data to be fetched, is passed to *sockargs*.

**uipc\_usrreq.c** The binding and connection of Unix domain sockets has been cleaned up so that *recvfrom* and *accept* get the address of the peer (if bound) rather than their own. The Unix-domain connection block records the bound address of a socket, not the address of the socket to which it is connected. For stream sockets, back pressure to implement flow control is now handled by adjusting the limits in the send buffer without overloading the normal count fields; the flow control information was moved to the connection block. Access rights are checked now when connecting; the connected-to socket must be writable by the caller, or the connection request is denied. In order to test one previously unused routine, the Unix domain stream support was modified to support the passage of access rights. Problems with access-rights garbage collection were also noted and fixed, and a count is kept of rights outstanding so that garbage collection is done only when needed. Garbage collection is triggered by socket shutdown now rather than file close; in 4.2BSD, it happened prematurely. The PRU\_SENSE *usrreq* entry, used by *stat*, has been added. It returns the write buffer size as the "blocksize," and generates a fake inode number and device for the benefit of those programs that use *fstat* information to determine whether file descriptors refer to the same file. Unimplemented requests have been carefully checked to see that they properly free mbufs when required and never otherwise. Larger buffers are allocated for both stream and datagram sockets. A number of minor bugs have been corrected: the back pointer from an inode to a socket needed to be cleared before release of the inode when detaching; sockets can only be bound once, rather than losing inodes; datagram sockets are correctly marked as connected and disconnected; several mbuf leaks were plugged. A serious problem was corrected in *unp\_drop*: it did not properly abort pending connections, with the result that closing a socket with unaccepted connections would cause an infinite loop trying to drop them.

#### 4.2. Changes in the virtual memory system

The virtual memory system in 4.3BSD is largely unchanged from 4.2BSD. The changes that have been made were in two areas: adapting the VM substem to larger physical memories, and optimization by simplifying many of the macros.

Many of the internal limits on the virtual memory system were imposed by the *cmap* structure. This structure was enlarged to increase those limits. The limit on physical memory has been changed from 8 megabytes to 64 megabytes, with expansion space provided for larger limits, and the limit of 15 mounted file systems has been changed to 255. The maximum file system size has been increased to 8 gigabytes, number of processes to 65536, and per-process size to 64 megabytes of data and 64 megabytes of stack. Configuration parameters and other segment size limits were converted from pages to bytes. Note that most of these are upper bounds; the default limits for these quantities are tuned for systems with 4-8 megabytes of physical memory. The process region sizes may be adjusted with kernel configuration file options; for example,

```
options    MAXDSIZ=33554432
```

increases the data segment to 32 megabytes. With no option, data segments receive a hard limit of

roughly 17Mb and a soft limit of 6Mb (that may be increased with the `cs limit` command).

The global clock page replacement algorithm used to have a single hand that was used both to mark and to reclaim memory. The first time that it encountered a page it would clear its reference bit. If the reference bit was still clear on its next pass across the page, it would reclaim the page. (On the VAX, the reference bit was simulated using the valid bit.) The use of a single hand does not work well with large physical memories as the time to complete a single revolution of the hand can take up to a minute or more. By the time the hand gets around to the marked pages, the information is usually no longer pertinent. During periods of sudden shortages, the page daemon will not be able to find any reclaimable pages until it has completed a full revolution. To alleviate this problem, the clock hand has been split into two separate hands. The front hand clears the reference bits, and the back hand follows a constant number of pages behind, reclaiming pages that have not been referenced since the front hand passed. While the code has been written in such a way as to allow the distance between the hands to be varied, we have not yet found any algorithms suitable for determining how to dynamically adjust this distance. The parameters determining the rate of page scan have also been updated to reflect larger configurations. The free memory threshold at which *pageout* begins was reduced from one-fourth of memory to 512K for machines with more than 2 megabytes of user memory. The scan rate is now independent of memory size instead of proportional to memory size.

The text table is now managed differently. Unused entries are treated as a cache, similar to the usage of the inode table. Entries with reference counts of 0 are placed in an LRU cache for potential reuse. In effect, all texts are "sticky," except that they are flushed after a period of disuse or overflow of the table. The sticky bit works as before, preventing entries from being freed and locking text files into the cache. The code to prevent modification of running texts was cleaned up by keeping a pointer to the text entry in the inode, allowing texts to be freed when unlinking files without linear searches.

The swap code was changed to handle errors a bit better (*swapout* doesn't do *swkills*, it just reflects errors to the caller for action there). During swapouts, interrupts are now blocked for less time after freeing the pages of the user structure and page tables (as explained by the old comment from *swapout*, "XXX hack memory interlock"), and this is now done only when swapping out the current process. The same situation existed in *exit*, but had not yet been protected by raised priority.

Various routines that took page numbers as arguments now take *cmap* pointers instead to reduce the number of conversions. These include *mlink*, *munlink*, *mlock*, *munlock*, and *mwait*. *Mlock* and *munlock* are generally used in their macro forms.

The remainder of the section details the other changes according to source file.

<b>vm_mem.c</b>	Low-level support for mapped files was removed, as the descriptor field in the page table entry was too small. Callers of <i>munhash</i> must block interrupts with <i>splmp</i> between checking for the presence of a block in the hash list and removing it with <i>munhash</i> in order to avoid reallocation of the page and a subsequent panic.
<b>vm_page.c</b>	When filling a page from the text file, <i>pagein</i> uses a new routine, <i>fodkluster</i> , to bring in additional pages that are contiguous in the filesystem. If errors occur while reading in text pages, no page-table change is propagated to other users of the shared image, allowing them to retry and notice the error if they attempt to use the same page. Virtual memory initialization code has been collected into <i>vmunit</i> , which adjusts swap interleaving to allow the configured size limits, set up the parameters for the clock algorithm, and set the initial virtual memory-related resource limits. The limit to resident-set size is set to the size of the available user memory. This change causes a single large process occupying most of memory to begin random page replacement as memory resources run short. Several races in <i>pagein</i> have been detected and fixed. Most of the <i>pageout</i> code was moved to <i>checkpage</i> in implementing the two-handed clock algorithm.
<b>vm_proc.c</b>	The <i>setjmp</i> in <i>procdup</i> was changed to <i>savectx</i> , which saves all registers, not just those needed to locate the others on the stack.



vm_pt.c	The <i>setjmp</i> call in <i>ptexpand</i> was changed to <i>savectx</i> to save all registers before initiating a swapout. <i>Vrelu</i> does an <i>splimp</i> before freeing user-structure pages if running on behalf of the current process. This had been done by <i>swapout</i> before, but not by <i>exit</i> .
vm_sched.c	The swap scheduler looks through the <i>allproc</i> list for processes to swap in or out. A call to <i>remrq</i> when swapping sleeping processes was unnecessary and was removed. If swapouts fail upon exhaustion of swap space, <i>sched</i> does not continue to attempt swapouts.
vm_subr.c	The <i>ptetov</i> function and the unused <i>vtopte</i> function were recoded without using the usual macros in order to fold the similar cases together.
vm_sw.c	The error returned by <i>swapon</i> when the device is not one of those configured was changed from <i>ENODEV</i> to <i>EINVAL</i> for accuracy. The search for the specified device begins with the first entry so that the error is correct ( <i>EBUSY</i> ) when attempting to enable the primary swap area.
vm_swap.c	The <i>swapout</i> routine now leaves any <i>swkill</i> to its caller. This avoids killing processes in a few situations. It uses <i>xdetach</i> instead of <i>xccdec</i> . Several unneeded <i>spl</i> 's were deleted.
vm_swp.c	The <i>swap</i> routine now consistently returns error status. <i>Physio</i> was modified to do scatter-gather I/O correctly.
vm_text.c	The text routines use a text free list as a cache of text images, resulting in numerous changes throughout this file. <i>Xccdec</i> now works only on locked text entries, and is replaced by <i>xdetach</i> for external callers. <i>Xumount</i> frees unused swap images from all devices when called with <i>NODEV</i> as argument. It is no longer necessary to search the text table to find any text associated with an inode in <i>xrele</i> , as the inode stores a pointer to any text entry mapping it. Statistics are gathered on the hit rate of the cache and its cost.

## 5. Machine specific support

The next several sections describe changes to the VAX-specific portion of the kernel whose sources reside in */sys/vax*.

### 5.1. Autoconfiguration

The data structures and top level of autoconfiguration have been generalized to support the VAX 8600 and machines whose main I/O busses are not similar to an SBI. The *percpu* structure has been broken into three structures. The *percpu* structure itself contains only the CPU type, an approximate value for the speed of the CPU, and a pointer to an array of I/O bus descriptions. Each of these, in turn, contain general information about one I/O bus that must be configured and a pointer to the private data for its configuration routine. The third new structure that has been defined describes the SBI and the other interconnects that emulate it. At boot time, *configure* calls *probeio* to configure the I/O bus(es). *Probeio* looks through the array of bus descriptions, indirecting to the correct routine to configure each bus. For the VAXen currently supported, the main bus is configured by either *probe\_Abus* (on the 8600 and 8650) or by *probenexi*, that is used on anything resembling an SBI. Multiple SBI adaptors on the 8600 are handled by multiple calls to *probenexi*. (Although the code has been tested with a second SBI, there were no adaptors installed on the second SBI.) This structure is easily extensible to other architectures using the BI bus, Q bus, or any combination of busses.

The CPU speed value is used to scale the *DELAY* macro so that autoconfiguration of old devices on faster CPU's will continue to work. The units are roughly millions of instructions per second (MIPS), with a value of 1 for the 780, although fractional values are not used. When multiple CPU's share the same CPU type, the largest value for any of them is used.

UNIBUS autoconfiguration has been modified to accommodate UNIBUS memory devices correctly. A new routine, *ubameminit*, is used to configure UNIBUS memory before probing other

devices, and is also used after a UNIBUS reset to remap these memory areas. The device probe or attach routines may then allocate and hold UNIBUS map registers without interfering with these devices.

## 5.2. Memory controller support

The introduction of the MS780-E memory controller for the VAX 780 made it necessary to configure the memory controller(s) on a VAX separately from the CPU. During autoconfiguration, the types of the memory controllers are recorded in an array. Memory error routines that must know the type of controller then use this information rather than the CPU type. The MS780-E controller is listed as two controllers, as each half reports errors independently. Both 1Mb and 4Mb boards using 64K and 256K dRAM chips are supported.

<b>Locore.c</b>	For <i>lint</i> 's sake, <i>Locore.c</i> has been updated to include the functions provided by <i>inline</i> and the new functions in <i>locore.s</i> .
<b>autoconf.c</b>	Most of the changes to autoconfiguration are described above. Other minor changes: UNIBUS controller probe routines are now passed an additional argument, a pointer to the <i>uba_ctlr</i> structure, and similarly device probe routines are passed a pointer to the <i>uba_device</i> structure. <i>Ubaaccess</i> and <i>nxaccess</i> were combined into a single routine to map I/O register areas. A logic error was corrected so that swap device sizes that were initialized from information in the machine configuration file are used unmodified. <i>Dumplo</i> is set at configuration time according to the sizes of the dump device and memory.
<b>conf.c</b>	Several new devices have been added and old entries have been deleted. A number of devices incorrectly set unused UNIBUS reset entries to <i>nodev</i> ; these were changed to <i>nulldev</i> . An entry was added for the new error log device. Additional device numbers have been reserved for local use.
<b>cons.h</b>	New definitions have been added for the 8600 console.
<b>cr1.h, cr1.c</b>	New files for the VAX 8600 console RL02 (our third RL02 driver!).
<b>flp.c</b>	It was discovered that not all VAXen that are not 780's are 750's; the console floppy driver for the 780 now checks for <i>cpu == 780</i> , not <i>cpu != 750</i> . An error causing the floppy to be locked in the busy state was corrected.
<b>genassym.c</b>	Several new structure offsets were needed by the assembly language routines.
<b>in_cksum.c</b>	It was discovered that the instruction used to clear the carry in the checksum loops did not actually clear carry. As the carry bit was always off when entering the checksum loop, this was never noticed.
<b>inline</b>	This directory contains the new <i>inline</i> program used to edit the assembly language output by the compiler.
<b>locore.s</b>	<p>The assembly language support for the kernel has a number of changes, some of which are VAX specific and some of which are needed on all machines. They are simply enumerated here without distinction.</p> <p>The <i>doadump</i> routine sometimes faulted because it changed the page table entry for the <i>rpb</i> without flushing the translation buffer. In order to reconfigure UNIBUS memory devices again after UNIBUS resets, <i>badaddr</i> was reimplemented without the need to modify the system control block. The machine check handler catches faults predicted by <i>badaddr</i>, cleans up and then returns to the error handler. The interrupt vectors have each been modified to count the number of interrupts from their respective devices, so that it is possible to account for software interrupts and UBA interrupts, and to determine which of several similar devices is generating unexpected interrupt loads. The <i>config</i> program generates the definitions for the indices into this interrupt count table. Software clock interrupts no longer call timer entries in the <i>dz</i> and <i>dh</i> drivers. The processing of network software interrupts has been reordered so that new interrupts requested during the protocol interrupt routine are likely to be</p>



handled before return from the software interrupt. Additional map entries were added to the network buffer and user page table page maps, as both use origin-1 indexing. The memory size limit and the offsets into the coremap are both obtained from *cmap.h* instead of inline constants. The signal trampoline code is all new and uses the *sigreturn* system call to reset signal masks and perform the *rei* to user mode. The initialization code for process 1, *icode*, was moved to this file to avoid hand assembly; it has been changed to exit instead of looping if */etc/init* cannot be executed, and to allow arguments to be passed to *init*. The routines that are called with *jsb* rather than *calls* use a new entry macro that allows them to be profiled if profiling is enabled.

Several new routines were added to move data from address space to address space a character string at a time; they are *copyinstr*, *copyoutstr*, and *copystr*. *Copyin* and *copyout* now receive their arguments in registers. *Setjmp* and *longjmp* are now similar to the user-level routines; *setjmp* saves the stack and frame pointers and PC only (all implemented in line), and *longjmp* unwinds the stack to recover the other registers. This optimizes the common case, *setjmp*, and allows the same semantics for register variables as for stack variables. For swaps and alternate returns using *u.u\_save*, however, all registers must be saved as in a context switch, and *savectx* is provided for that purpose.

Redundant context switches were caused by two bugs in *switch*. First, *switch* cleared *runrun* before entering the idle loop. Once an interrupt caused a *wakeup*, *runrun* would be set, requesting another context switch at system call exit. Also, the use of the VAX AST mechanism caused a similar problem, posting AST's to one process that would then *switch* (or might already be in the idle loop), only to catch the AST after being rescheduled and completing its system service. The AST is no longer marked in the process control block and is cancelled during the context switch. The idle loop has been separated from *switch* for profiling.

The *startup* code to calculate the core map size and the limit to the buffer cache's virtual memory allocation was corrected and reworked. The number of buffer pages was reduced for larger memories (10% of the first 2 Mb of physical memory is used for buffers, as before, and 5% thereafter). The default number of buffers or buffer pages may be overridden with configuration-file options. If the number of buffers must be reduced to fit the system page table, a warning message is printed. Buffers are allocated after all of the fully dense data structures, allowing the other tables allocated at boot time to be mapped by the identity map once again. The new signal stack call and return mechanisms are implemented here by *sendsig* and *sigreturn*; *sigcleanup* remains for compatibility with 4.2BSD's *longjmp*. There are a number of modifications for the VAX 8600, particularly in the machine check and memory error handlers and in the use of the console flags. On the VAX-11/750 more translation-buffer parity faults are considered recoverable. The *reboot* routine flushes the text cache before initiating the filesystem update, and may wait longer for the update to complete. The time-of-day register is set, as any earlier time adjustments are not reflected there yet. The *microtime* function was completed and is now used; it is careful not to allow time to appear to reverse during time corrections. An *initcpu* routine was added to enable caches, floating point accelerators, etc.

The file *vax/param.h* was renamed to avoid ambiguity when including "*param.h*".

This new file contains the checksum code for the Xerox NS network protocols.

The *aston()* and *astoff()* macros no longer set an AST in the process control block (see *locore.s*).

The *pg\_blkno* field was increased to 24 bits to correspond with the *cmap* structure; the *pg\_fileno* field was reduced to a single bit, as it no longer contains a file descriptor.

machdep.c

machparam.h

ns\_cksum.c

pcb.h

pte.h



<code>swappgeneric.c</code>	<i>Dumpdev</i> and <i>argdev</i> are initialized to <i>NODEV</i> , preventing accidents should they be used before configuration completes. <i>DEL</i> is now recognized as an erase character by the kernel <i>gets</i> .
<code>tmscp.h</code>	A new file which contains definitions for the Tape Mass Storage Control Protocol.
<code>trap.c</code>	Syscall 63 is no longer reserved by <i>syscall</i> for out-of-range calls. In order to make <i>wait3</i> restartable, <i>syscall</i> must not clear the carry bit in the program status longword before beginning a system call, but only after successful completion.
<code>tu.c</code>	There were several important fixes in the console TU58 driver.
<code>vm_machdep.c</code>	The <i>chksize</i> routine requires an additional argument, allowing it to check data size and bss growth separately without overflow.
<code>vmparam.h</code>	The limits to user process virtual memory allow nondefault values to be defined by configuration file options. The definition of <i>DMMAX</i> here now defines only the maximum value; it will be reduced according to the definition of <i>MAXDSIZ</i> . The space allocated to user page tables was increased substantially. The free-memory threshold at which <i>pageout</i> begins was changed to be at most 512K.

## 6. Network

There have been many changes in the kernel network support. A major change is the addition of the Xerox NS protocols. During the course of the integration of a second major protocol family to the kernel, a number of Internet dependencies were removed from common network code, and structural changes were made to accommodate multiple protocol and address families simultaneously. In addition, there were a large number of bug fixes and other cleanups in the general networking code and in the Internet protocols. The skeletal support for PUP that was in 4.2BSD has been removed.

The link layer drivers were changed to save an indication of the incoming interface with each packet received, and this information was made available to the protocol layer. There were several problems that could be corrected by taking advantage of this change. The IMP code needed to save error packets for software interrupt-level processing in order to fix a race condition, but it needed to know which interface had received the packet when decoding the addresses. ICMP needed this information to support information requests and (newly added) network mask requests properly, as these request information about a specific network. IP was able to take advantage of this change to implement redirect generation when the incoming and outgoing interfaces are the same.

### 6.1. Network common code

The changes in the common support routines for networking, located in */sys/net*, are described here.

<code>if_arp.h</code>	This new file contains the definitions for the Address Resolution Protocol (ARP) that are independent of the protocols using ARP.
<code>if.c</code>	Most of the <i>if_ifwith*</i> functions that returned pointers to <i>ifnet</i> structures were converted to <i>ifa_with*</i> equivalents that return pointers to <i>ifaddr</i> structures. The old <i>if_onnetof</i> function is no longer provided, as there is no concept of network number that is independent of address family. A new routine, <i>ifa_ifwithdstaddr</i> , is provided for use with point-to-point interfaces. Interface <i>ioctl</i> s that set interface addresses are now passed to the appropriate protocol using the <i>PRU_CONTROL</i> request of the <i>pr_usrreq</i> entry. Additional <i>ioctl</i> operations were added to get and set interface metrics and to manipulate the ARP table (see <i>netinet/if_ether.c</i> ).
<code>if.h</code>	In 4.2BSD, the per-interface structure <i>ifnet</i> held the address of the interface, as well as the host and network numbers. These have all been moved into a new structure, <i>ifaddr</i> , that is managed by the address family. The <i>ifnet</i> structure for an interface includes a pointer to a linked list of addresses for the interface. The <i>IFF_ROUTE</i> flag was also removed. The software loopback interface is distinguished with a new flag. Each interface now has a routing metric that is stored by the kernel but only

- interpreted by user-level routing processes. Additional interface *ioctl* operations allow the metric or the broadcast address to be read or set. When received packets are passed to the receiving protocol, they include a reference to the incoming interface; a variant of the *IF\_DEQUEUE* macro, *IF\_DEQUEUEIFP*, dequeues a packet and extracts the information about the receiving interface.
- if\_loop.c** The software loopback driver now supports Xerox NS and Internet protocols. It was modified to provide information on the incoming interface to the receiving protocol. The loopback driver's address(es) must now be set with *ifconfig*.
- if\_sl.c** This file was added to support a customized line discipline for the use of an asynchronous serial line as a network interface. Until the encapsulation is changed the interface supports only IP traffic.
- raw\_cb.c** Raw sockets record the socket's protocol number and address family in a *sockproto* structure in the raw connection block. This allows a wildcard raw protocol entry to support raw sockets using any single protocol.
- raw\_cb.h** A *sockproto* description and a hook for protocol-specific options were added to the raw protocol control block.
- raw\_usrreq.c** A bug was fixed that caused received packet return addresses to be corrupted periodically; an mbuf was being used after it was freed. Routing is no longer done here, although the raw socket protocol control block includes a routing entry for use by the transport protocol. The *SO\_DONTROUTE* flag now works correctly with raw sockets.
- route.c** The routing algorithm was changed to use the first route found in the table instead of the one with the lowest use count. This reduces routing overhead and makes response more predictable. The load-sharing effect of the old algorithm was minimal under most circumstances. Several races were fixed. The hash indexes have been declared as unsigned; negative indices worked for the network route hash table but not for the host hash table. (This fix was included on most 4.2BSD tapes.) New routes are placed at the front of the hash chains instead of at the end. The redirect handling is more robust; redirects are only accepted from the current router, and are not used if the new gateway is the local host. The route allocated while checking a redirect is freed even if the redirect is disbelieved. Host redirects cause a new route to be created if the previous route was to the network. Routes created dynamically by redirects are marked as such. When adding new routes, the gateway address is checked against the addresses of point-to-point links for exact matches before using another interface on the appropriate network. *Rtinit* takes arguments for flags and operation separately, allowing point-to-point interfaces to delete old routes.
- route.h** The size of the routing hash table has been changed to a power of two, allowing unsigned modulus operations to be performed with a mask. The size of the table is expanded if the *GATEWAY* option is configured.

## 7. Internet network protocols

There are numerous bug fixes and extensions in the Internet protocol support (*/sys/netinet*). This section describes some of the more important changes with very little detail. As many of the changes span several source files, and as it is very difficult to merge this code with earlier versions of these protocols, it is strongly recommended that the 4.3BSD network be adopted intact, with local hacks merged into it only if necessary.

### 7.1. Internet common code

By far, the most important change in IP and the shared Internet support layer is the addition of subnetwork addressing. This facility is used (and required) by a number of large university and other networks that include multiple physical networks as well as connections with the DARPA Internet. Subnet support allows a collection of interconnected local networks to share a single network number,

hiding the complexity of the local environment and routing from external hosts and gateways. The subnet support in 4.3BSD conforms with the Internet standard for subnet addressing, RFC-950. For each network interface, a network mask is set along with the address. This mask determines which portion of the address is the network number, including the subnet, and by default is set according to the network class (A, B, or C, with 8, 16, or 24 bits of network part, respectively). Within a subnetted network each subnet appears as a distinct network; externally, the entire network appears to be a single entity.

Another important change in IP addressing is a change to the default IP broadcast address. The default broadcast address is the address with a host part of all ones (using the definition `INADDR_BROADCAST`), in conformance with RFC-919. In 4.2BSD, the broadcast address was the address with a host part of all zeros (`INADDR_ANY`). To facilitate the conversion process, and to help avoid breaking networks with forwarded broadcasts, 4.3BSD allows the broadcast address to be set for each interface. IP recognizes and accepts network broadcasts as well as subnet broadcasts when subnets are enabled. Such broadcasts normally originate from hosts that do not know about subnets. IP also accepts old-style (4.2) broadcasts using a host part of all zeros, either as a network or subnet broadcast. An address of all ones is recognized as "broadcast on this network," and an address of all zeros is accepted as well. The latter two are sometimes used in broadcast information requests or network mask requests in the course of starting a diskless workstation. ICMP includes support for the Network Mask Request and Response. A new routine, *in\_broadcast*, was added for the use of link layer output routines to determine whether an IP packet should be broadcast.

Network numbers are now stored and used unshifted to minimize conversions and reduce the overhead associated with comparisons. 4.2BSD shifted network numbers to the low-order part of the word. The structure defining Internet addresses no longer includes the old IMP-host fields, but only a featureless 32-bit address.

- |                 |   |
|-----------------|---|
| <b>in.h</b>     | The definitions of Internet port numbers in this file were deleted, as they have been superceded by the <i>getservicebyname</i> interface. A definition was added for the single option at the IP level accessible through <i>setsockopt</i> , <code>IP_OPTIONS</code> .  |
| <b>in_pcb.h</b> | The Internet protocol control block includes a pointer to an optional mbuf containing IP options.   |
| <b>in_var.h</b> | This new header file contains the declaration of the Internet variety of the per-interface address information. The <i>in_ifaddr</i> structure includes the network, subnet, network mask and broadcast information.  |
| <b>in.c</b>     | The <i>if_*</i> routines which manipulate Internet addresses were renamed to <i>in_*</i> . <i>in_netof</i> and <i>in_lnaof</i> check whether the address is for a directly-connected network, and if so they use the local network mask to return the subnet/net and host portions, respectively. <i>in_localaddr</i> determines whether an address corresponds to a directly-connected network. By default, this includes any subnet of a local network; a configuration option, <code>SUBNETSARELOCAL=0</code> , changes this to return true only for a directly-connected subnet or non-subnetted network. Interface <i>ioctl</i> s that get or set addresses or related status information are forwarded to <i>in_control</i> , which implements them. <i>in_iaonnetof</i> replaces <i>if_ifonnetof</i> for Internet addresses only.  |
| <b>in_pcb.c</b> | The destination address of a <i>connect</i> may be given as <code>INADDR_ANY</code> (0) as a short-hand notation for "this host." This simplifies the process of connecting to local servers such as the name-domain server that translates host names to addresses. Also, the short-hand address <code>INADDR_BROADCAST</code> is converted to the broadcast address for the primary local network; it fails if that network is incapable of broadcast. The source address for a connection or datagram is selected according to the outgoing interface; the initial route is allocated at this time and stored in the protocol control block, so that it may be used again when actually sending the packet(s). The <i>in_pcbnotify</i> routine was generalized to apply any function and/or report an error to all connections to a destination; it is used to notify connections of routing changes and other non-error situations as well as errors. New entries have been added to this |



level to invalidate cached routes when routing changes occur, as well as to report possible routing failures detected by higher levels.

**in\_proto.c** The protocol switch table for Internet protocols includes entries for the *ctloutput* routines. ICMP may be used with raw sockets. A raw wildcard entry allows raw sockets to use any protocol not already implemented in the kernel (e.g., EGP).

## 7.2. IP

Support was added for IP source routing and other IP options (partly derived from BBN's implementation). On output, IP options such as strict or loose source route and record may be set by a client process using TCP, UDP or raw IP sockets. IP properly updates source-route and record-route options when forwarding (and leaves them in the packet, unlike 4.2 which stripped them out after updating). IP input preserves any source-routing information in an incoming packet and passes it up to the receiving protocol upon request, reversing it and arranging it in the same way as user-supplied options. Both TCP and ICMP retrieve incoming source routes for use in replies. Most of the option-handling code has been converted to use *bcopy* instead of structure assignments when copying addresses, as the alignment in the incoming packet may not be correct for the host. This is not required on the VAX, but is needed on most other machines running 4.2BSD.

**ip.h** The IP time-to-live field is decremented by one when forwarding; in 4.2BSD this value was five.

**ip\_var.h** Data structures and definitions were added for storing IP options. New fields have been added to the structure containing IP statistics.

**ip\_input.c** The changes to save and present incoming IP source-routing information to higher level protocols are in this file. The identity of the interface that received the packet is also determined by *ip\_input* and passed to the next protocol receiving the packet. To avoid using uninitialized data structures, IP must not begin receiving packets until at least one Internet address has been set. A bug in the reassembly of IP packets with options has been corrected. Machines with only a single network interface (in addition to the loopback interface) no longer attempt to forward received IP packets that are not destined for them; they also do not respond with ICMP errors unless configured with the GATEWAY option. This change prevents large increases in network activity which used to result when an IP packet that was broadcast was not understood as a broadcast. A one-element route cache was added to the IP forwarding routine. When a packet is forwarded using the same interface on which it arrived, if the source host is on the directly-attached network, an ICMP redirect is sent to the source. If the route used for forwarding was a route to a host or a route to a subnet, a host redirect is used, otherwise a network redirect is sent. The generation of redirects may be disabled by a configuration option, IPSENDREDIRECTS=0. More statistics are collected, in particular on traffic and fragmentation. The *ip\_ctlinput* routine was moved to each of the upper-level protocols, as they each have somewhat different requirements.

**ip\_output.c** The IP output routine manages a cached route in the protocol control block for each TCP, UDP or raw IP socket. If the destination has changed, the route has been marked down, or the route was freed because of a routing change, a new route is obtained. The route is not used if the IP\_ROUTETOIF (aka SO\_DONTROUTE or MSG\_DONTROUTE) option is present. Preformed IP options passed to *ip\_output* are inserted, changing the destination address as required. The *ip\_ctloutput* routine allows options to be set for an individual socket, validating and internalizing them as appropriate.

**raw\_ip.c** The type-of-service and offset fields in the IP header are set to zero on output. The SO\_DONTROUTE flag is handled properly.

### 7.3. ICMP

There have been numerous fixes and corrections to ICMP. Length calculations have been corrected, allowing most ICMP packet lengths to be received and allowing errors to be sent about smaller input packets. ICMP now uses information about the interface on which a message was received to determine the correct source address on returned error packets and replies to information requests. Support was added for the Network Mask Request. Responses to source-routed requests use the reversed source route for the return trip. Timestamps are created with *microtime*, allowing 1-millisecond resolution. The *icmp\_error* routine is capable of sending ICMP redirects. When processing network redirects, the returned source address is converted to a network address before passing it to the routing redirect handler. The translation of ICMP errors to Unix error returns was updated.

### 7.4. TCP

In addition to bug fixes, several performance changes have been made to TCP. Several of these address overall network performance and congestion avoidance, while others address performance of an individual connection. The most important changes concern the TCP send policy. First, the sender silly-window syndrome avoidance strategy was fixed. In 4.2BSD, the amount that could be sent was compared to the offered window, and thus small amounts could still be sent if the receiver offered a silly window. Once this was fixed, there were problems with peers that never offered windows large enough for a maximum segment, or at least 512 bytes (e.g., the peer is a TAC or an IBM PC). Code was then added to maintain estimates of the peer's receive and send buffer sizes. The send policy will now send if the offered window is at least one-half of the receiver's buffer, as well as when the window is at least a full-sized segment. (When the window is large enough for all data that is queued, the data will also be sent.) The send buffer size estimate is not yet used, but is desired for a new delayed-acknowledgement scheme that has yet to be tested. Another problem that was exposed when the silly-window avoidance was fixed was that the persist code didn't expect to be used with a non-zero window. The persist now lasts only until the first timeout, at which time a packet is sent of the largest size allowed by the window. If this packet is not acknowledged, the output routine must begin retransmission rather than returning to the persist state.

Another change related to the send policy is a strategy designed to minimize the number of small packets outstanding on slow links. This is an implementation of an algorithm proposed by John Nagle in RFC-896. The algorithm is very simple: when there is outstanding, unacknowledged data pending on a connection, new data are not sent unless they fill a maximum-sized segment. This allows bulk data transfers to proceed, but causes small-packet traffic such as remote login to bundle together data received during a single round-trip time. On high-bandwidth, low-delay networks such as a local Ethernet, this change seldom causes delay, but over slow links or across the Internet, the number of small packets can be reduced considerably. This algorithm does interact poorly with one type of usage, however, as demonstrated by the X window system. When small packets are sent in a stream, such as when doing rubber-banding to position a new window, and when no echo or other acknowledgement is being received from the other end of the connection, the round-trip delay becomes as large as the delayed-acknowledgement timer on the remote end. For such clients, a TCP option may be set with *setsockopt* to defeat this part of the send policy.

For bulk-data transfers, the largest single change to improve performance is to increase the size of the send and receive buffers. The default buffer size in 4.3BSD is 4096 bytes, double the value in 4.2BSD. These values allow more outstanding data and reduce the amount of time waiting for a window update from the receiver. They also improve the utility of the delayed-acknowledgement strategy. The delayed acknowledgment strategy withholds acknowledgements until a window update would uncover at least 35% of the window; in 4.2BSD, with 1024-byte packets on an Ethernet and 2048-byte windows, this took only a single packet. With 4096-byte windows, up to 50% of the acknowledgements may be avoided.

The use of larger buffers might cause problems when bulk-data transfers must traverse several networks and gateways with limited buffering capacity. The source-quench ICMP message was provided to allow gateways in such circumstances to cause source hosts to slow their rate of packet



injection into the network. While 4.2BSD ignored such messages, the 4.3BSD TCP includes a mechanism for throttling back the sender when a source quench is received. This is done by creating an artificially small window (one which is 80% of the outstanding data at the time the quench is received, but no less than one segment). This artificial congestion window is slowly opened as acknowledgements are received. The result under most circumstances is a slow fluctuation around the buffering limit of the intermediate gateways, depending on the other traffic flowing at the same time.

A final set of changes designed to improve network throughput concerns the retransmission policy. The retransmission timer is set according to the current round-trip time estimate. Unfortunately, the round-trip timing code in 4.2BSD had several bugs which caused retransmissions to begin much too early. These bugs in round trip timing have been corrected. Also, the retransmission code has been tuned, using a faster backoff after the first retransmission. On an initial connection request where there is no round-trip time estimate, a much more conservative policy is used. When a slow link intervenes between the sender and the destination, this policy avoids queuing large numbers of retransmitted connection requests before a reply can be received. It also avoids saturation when the destination host is down or nonexistent. During a connection, when the retransmission timer expires, only a single packet is sent. When only a single packet has been lost, this avoids resending data that was successfully received; when a host has gone down or become unreachable, it avoids sending multiple packets at each timeout. Once another acknowledgement is received, the transmission policy returns to normal.

4.2BSD offered a maximum receive segment size of 1024 for all connections, and accepted such offers whenever made. However, that size was especially poor for the Arpanet and other 1822-based IMP networks (sorry, make that PSN networks) where the maximum packet size is 1007 bytes. This was compounded by a bug in the LH/DH driver that did not allow space for an end-of-packet bit in the receive buffer, and thus maximum size packets that were received were split across buffers. This, in turn, aggravated a hardware problem causing small packets following a segmented packet to be concatenated with the previous packet. The result of this set of conditions was that performance across the Arpanet was sometimes abominably slow. The maximum size segment selected by 4.3BSD is chosen according to the destination and the interface to be used. The segment size chosen is somewhat less than the maximum transmission unit of the outgoing interface. If the destination is not local, the segment size is a convenient small size near the default maximum size (512 bytes). This value is both the maximum segment size offered to the sender by the receive side, and the maximum size segment that will be sent. Of course, the send size is also limited to be no more than the receiver has indicated it is willing to receive.

The initial sequence number prototype for TCP is now incremented much more quickly; this has exposed two bugs. Both the window-update receiving code and the urgent data receiving code compared sequence numbers to 0 the first time they were called on a connection. This fails if the initial sequence number has wrapped around to negative numbers. Both are now initialized when the connection is set up. This still remains a problem in maintaining compatibility with 4.2BSD systems; thus an option, TCP\_COMPAT\_42, was added to avoid using such sequence numbers until 4.2 systems have been upgraded.

Additional changes in TCP are listed by source file:

**tcp\_input.c** The common case of TCP data input, the arrival of the next expected data segment with an empty reassembly queue, was made into a simplified macro for efficiency. *Tcp\_input* was modified to know when it needed to call the output side, reducing unnecessary tests for most acknowledgement-only packets. The receive window size calculation on input was modified to avoid shrinking the offered window; this change was needed due to a change in input data packaging by the link layer. A bug in handling TCP packets received with both data and options (that are not supposed to be used) has been corrected. If data is received on a connection after the process has closed, the other end is sent a reset, preventing connections from hanging in CLOSE\_WAIT on one end and FIN\_WAIT\_2 on the other. (4.2BSD contained code to do this, but it was never executed because such input packets had already been dropped as being outside of the receive window.) A timer is now started upon

entering `FIN_WAIT_2` state if the local user has closed, closing the connection if the final `FIN` is not received within a reasonable time. Half-open connections are now reset more reliably; there were circumstances under which one end could be rebooted, and new connection requests that used the same port number might not receive a reset. The urgent-data code was modified to remember which data had already been read by the user, avoiding possible confusion if two urgent-data signals were received close together. Another change was made specifically for connections with a TAC. The TAC doesn't fill in the window field on its initial packet (`SYN`), and the apparent window is random. There is some question as to the validity of the window field if the packet does not have `ACK` set, and therefore TCP was changed to ignore the window information on those packets.

- tcp\_output.c** The advertised window is never allowed to shrink, in correspondence with the earlier change in the input handler. The retransmit code was changed to check for shrinking windows, updating the connection state rather than timing out while waiting for acknowledgement. The modifications to the send policy described above are largely within this file.
- tcp\_timer.c** The timer routines were changed to allow a longer wait for acknowledgements. (TCP would generally time out before the routing protocol had changed routes.)

## 7.5. UDP

An error in the checksumming of output UDP packets was corrected. Checksums are now checked by default, unless the `COMPAT_42` configuration option is specified; it is provided to allow communication with the 4.2BSD UDP implementation, which generates incorrect checksums. When UDP datagrams are received for a port at which no process is listening, ICMP unreachable messages are sent in response unless the input packet was a broadcast. The size of the receive buffer was increased, as several large datagrams and their attached addresses could otherwise fill the buffer. The time-to-live of output datagrams was reduced from 255 to 30. UDP uses its own *ctlinput* routine for handling of ICMP errors, so that errors may be reported to the sender without closing the socket.

## 7.6. Address Resolution Protocol

The address resolution protocol has been generalized somewhat. It was specific for IP on 10 Mb/s Ethernet; it now handles multiple protocols on 10 Mb/s Ethernet and could easily be adapted to other hardware as well. This change was made while adding ARP resolution of trailer protocol addresses. Hosts desiring to receive trailer encapsulations must now indicate that by the use of ARP. This allows trailers to be used between cooperating 4.3 machines while using non-trailer encapsulations with other hosts. The negotiation need not be symmetrical: a VAX may request trailers, for example, and a SUN may note this and send trailer packets to the VAX without itself requesting trailers. This change requires modifications to the 10 Mb/s Ethernet drivers, which must provide an additional argument to *arpresolve*, a pointer for the additional return value indicating whether trailer encapsulations may be sent. With this change, the `IFF_NOTRAILERS` flag on each interface is interpreted to mean that trailers should not be requested. Modifications to ARP from SUN Microsystems add *ioctl* operations to examine and modify entries in the ARP address translation table, and to allow ARP translations to be "published." When future requests are received for Ethernet address translations, if the translation is in the table and is marked as published, they will be answered for that host. Those modifications superceded the "oldmap" algorithmic translation from IP addresses, which has been removed. Packets are not forwarded to the loopback interface if it is not marked up, and a bug causing an mbuf to be freed twice if the loopback output fails was corrected. ARP complains if a host lists the broadcast address as its Ethernet address. The ARP tables were enlarged to reflect larger network configurations now in use. A new function for use in driver messages, *ether\_sprintf*, formats a 48-bit Ethernet address and returns a pointer to the resulting string.

### 7.7. IMP support

The support facilities for connections to an 1822 (or X.25) IMP port (/sys/netimp) have had several bug fixes and one extension. Unit numbers are now checked more carefully during autoconfiguration. Code from BRL was installed to support class B and C networks. Error packets received from the IMP such as Host Dead are queued in the interrupt handler for reprocessing from a software interrupt, avoiding state transitions in the protocols at priorities above *splnet*. The host-dead timer is no longer restarted when attempting new output, as a persistent sender could otherwise prevent new output from being attempted once a host was reported down. The network number is always taken from the address configured for the interface at boot time; network 10 is no longer assumed. A timer is used to prevent blocking if RFNM messages from the IMP are lost. A race was fixed when freeing mbufs containing host table entries, as the mbuf had been used after it was freed.

## 8. Xerox Network Systems Protocols

4.3BSD now supports some of the Xerox NS protocols. The kernel will allow the user to send or receive IDP datagrams directly, or establish a Sequenced Packet connection. It will generate Error Protocol packets when necessary, and may close user connections if this is the appropriate action on receipt of such packets. It will respond to Echo Protocol requests. The Routing Information Protocol is executed by a user level process, and sufficient access has been left for other protocols to be implemented using IDP datagrams. It would be possible to set the additional fields required for the Packet Exchange format at user level, to provide a daemon to respond to time-of-day requests, or conduct an expanding ring broadcast to discover clearinghouses.

Wherever possible, the algorithms and data structures parallel those used in Internet protocol support, so that little extra effort should be required to maintain the NS protocols. There has not yet been much effort at tuning.

### 8.1. Naming

A machine running 4.3 is allowed to have only one six-byte NS host address, but is permitted to be on several networks. As in the Internet case, an address of all zeros may be used to bind the host address for an offered service. Unlike the Internet case, an address of all zeros cannot be used to contact a service on the same machine. (This should be changed.)

There is only one name space of port numbers, as opposed to the Internet case where each protocol has its own port space.

Several point-to-point connections can share the same network number. The destination of a point-to-point connection can have a different network number from the local end.

The files *ns.h*, *ns\_pcb.h*, *ns.c*, *ns\_pcb.c* and *ns\_proto.c* are direct translations of similarly named files in the *netinet* directory. *Ns\_pcbnotify* differs a little from *in\_pcbnotify* in that it takes an extra parameter which it will pass to the "notification" routine argument indirectly, by stuffing it in each NS control block selected.

This header file *ns\_if.h* contains the declaration of the NS variety of the per-interface address information, like *netinet/in\_var.h*.

### 8.2. Encapsulations

The stipulation that each host is allowed exactly one 6 byte address implies that each 10 Mb/s Ethernet interface other than the first will need to reprogram its physical address. All the 10 Mb/s Ethernet drivers supplied with 4.3BSD perform this. The 3 Mb/s Ethernet driver does not perform any address resolution, but uses the 6th byte of the NS host address as a PUP host number, making it largely incompatible with altos running XNS. In a system with both 3 Mb/s and 10 Mb/s Ethernets, one should configure the 3 Mb/s network first.

The file *ns\_ip.c* contains code providing a mechanism for sending XNS packets over any medium supporting IP datagrams. It builds objects that look like point-to-point interfaces from the point of view of NS, and a protocol from the point of view of IP. Each of these pseudo interface



structures has extra IP data at the end (a route, source and destination), and fits exactly into an mbuf. If the *ifnet* structure grows any larger, the extra data will have to be put in a separate mbuf, or the whole scheme will have to be reworked more rationally.

### 8.3. Datagrams

The files *ns\_input.c* and *ns\_output.c* contain the base level routines which interact with network interface drivers. There is a kernel variable *idp\_cksum*, which can be used to defeat checksums for all packets. (There ought to be an option per socket to do this). The NS output routine manages a cached route in the protocol control block of each socket. If the destination has changed, the route has been marked down, or the route was freed because of a routing change, a new route is obtained. The route is not used if the NS\_ROUTETOIF (aka SO\_DONTRROUTE or MSG\_DONTRROUTE) option is present.

The files *idp.h*, *idp\_var.h*, and *idp\_usrreq.c* are the analogues of *udp.h*, *udp\_var.h*, and *udp\_usrreq.c*.

### 8.4. Error and Echo protocols

Routines for processing incoming error protocol packets are in *ns\_error.c*. They call *ctlinput* routines for IDP and SPP to maintain structural similarity to the Internet implementation. The kernel will generate error messages indicating lack of a listener at a port, incorrectly received checksum, or that a packet was thrown away due to insufficient resources at the recipient (buffer full). The echo protocol is handled as a special case. If there is no listener at port number 2, then the routine that generates the "no listener" error message will inspect the packet to see if it was an echo request, and if so, will echo it. Thus, the user is free to construct his own echoing daemon if he so chooses.

### 8.5. Sequenced Packet Protocol

In general, this code employs the Internet TCP algorithms where possible. By default, a three-way handshake is used in establishing connections. There is a compile time option to employ the minimal two way handshake. Incoming connections may multiplexed by source machine and port, as in the Internet case. It will switch over ports when establishing connections if requested to do so.

The retransmission timing and strategies are much like those of TCP, though recent performance enhancements have not yet migrated here. There has not yet been much opportunity to tune this implementation. The code is intended to generate keep-alive packets, though there is some evidence this isn't working yet. The TCP source-quench strategy hasn't been added either. The default nominal packet size is 576 bytes, and the default amount of buffering is 2048. It is possible to raise both by setting appropriate socket options.

## 9. VAX Network Interface drivers

Most of the changes in the network interfaces follow common patterns that are summarized in categories. In addition, there are a number of bug fixes. The change that was made universally to the interface handlers was to remove the *ioctl* routines that set the interface address and flags, replacing them by simpler routines that merely initialize the hardware if this has not already been done. Several of the drivers notice when the IFF\_UP flag is cleared and perform a hardware reset, then reinitialize the interface when IFF\_UP is set again. This allows interfaces to be turned off, and also provides a mechanism to reset devices that have lost interrupts or otherwise stopped functioning. The handling of the other interface flags has been made more consistent. IFF\_RUNNING is used uniformly to indicate that UNIBUS resources have been allocated and that the board has been initialized. The reset routines clear this flag before reinitializing so that both operations will be repeated.

### 9.1. Interface UNIBUS support

The UNIBUS common support routines for network interfaces have been modified to support multiple transmit and receive buffers per device. A set of macros provide a nearly-compatible interface for devices using a single buffer of each type. When placing received packets into mbufs,

*if\_ubaget* prepends a pointer to the receiving interface to the data; this requires that the interface pointer be passed to *if\_ubaget* or *if\_rubaget* as an additional argument. When removing the trailer header from the front of a packet, interface receive routines must move the interface pointer which precedes the header; see one of the existing drivers for an example. When received data is larger than half of an mbuf cluster, the data will be placed in an mbuf cluster rather than a chain of small mbufs. Similarly, in *if\_ubaput*, clusters may be remapped instead of copied if they are at least one-half full and are the last mbuf of the chain. For devices like the DEC DEUNA that wish to perform receive operations on a transmit buffer, the transmit buffers are marked. Receive operations from transmit buffers force page mapping to be consistent before attempting to read data or swap pages from them.

## 9.2. 10 Mb/s Ethernet

The 10Mb/s Ethernet handlers have been modified to use the new ARP interfaces. They no longer use *arpattach*, and the call to *arpresolve* contains an additional argument for a second return, a boolean for the use of trailer encapsulations. Input and output functions were augmented to handle NS IDP packets. For hosts using Xerox NS with multiple interfaces, the drivers are able to reprogram the physical address on each board so that all interfaces use the address of the first configured interface. The hardware Ethernet addresses are printed during autoconfiguration.

## 9.3. Changes specific to individual drivers

<i>if_acc.c</i>	An additional word was added to the input buffer to allow space for the end-of-message bit on a maximum-sized message without segmentation. This avoids a hardware problem that sometimes causes the next packet to be concatenated with the end-of-message segment.
<i>if_ddn.c</i>	A new driver from ACC DDN Standard mode X.25 IMP interface.
<i>if_de.c</i>	A new driver for the DEC DEUNA 10 Mb/s Ethernet controller. The hardware is reset when <i>ifconfig</i> ed down and reinitialized when marked up again.
<i>if_dmc.c</i>	The DMC-11/DMR-11 driver has been made much more robust. It now uses multiple transmit and receive buffers. A link-layer encapsulation is used to indicate the type of the packet; this driver is thus incompatible with the 4.2BSD DMC driver. (The driver is, however, compatible with current ULTRIX drivers.)
<i>if_ec.c</i>	The handler for the 3Com 10 Mb/s Ethernet controller is now able to support multiple units. The address of the UNIBUS memory is taken from the flags in the configuration file; note that address 0 is still the default. The UNIBUS memory is configured in a separate memory-probe routine that is called during autoconfiguration and after a UNIBUS reset. This allows the 3Com interface reset to work correctly. The collision backoff algorithm was corrected so that the maximum backoff is within the specification, rather than waiting seconds after numerous collisions. The private <i>ecget</i> and <i>ecput</i> routines were modified to correspond with the <i>if_uba</i> routines. The hardware is reset when <i>ifconfig</i> ed down and reinitialized when marked up again.
<i>if_en.c</i>	The 3 Mb/s Experimental Ethernet driver now supports NS IDP packets, using a simple algorithmic conversion of host to Ethernet addresses. The <i>enswab</i> function was corrected.
<i>if_ex.c</i>	A new driver for the Excelan 204 10 Mb/s Ethernet controller, used as a link-layer interface.
<i>if_hdh.c</i>	A new driver for the ACC HDH IMP interface.
<i>if_hy.c</i>	A new version of the Hyperchannel driver from Tektronix was installed. It is untested with 4.3BSD.
<i>if_il.c</i>	The Interlan 1010 and 1010A driver now resets the interface and checks the result of hardware diagnostics when initializing the board. The hardware is reset when <i>ifconfig</i> ed down and reinitialized when marked up again.

- if\_ix.c**      A new driver for using the Interlan NP100 10 Mb/s Ethernet controller as a link-level interface.
- if\_uba.c**      In addition to the major changes in UNIBUS support functions, there were several bug fixes made. Interfaces with no link-level header are set up properly. A variable was reused incorrectly in *if\_wubaput*, and this has been corrected.
- if\_vv.c**      The driver for the Proteon proNET has been reworked in several areas. The elaborate error handling code had several problems and was simplified considerably. The driver includes support for both the 10 Mb/s and 80 Mb/s rings. The byte ordering of the trailer fields was corrected; this makes the trailer format incompatible with the 4.2BSD driver.

## 10. VAX MASSBUS device drivers

This section documents the modifications in the drivers for devices on the VAX MASSBUS, with sources in */sys/vaxmba*, as well as general changes made to all disk and tape drivers.

### 10.1. General changes in disk drivers

Most of the disk drivers' strategy routines were changed to report an end-of-file when attempting to read the first block after the end of a partition. Distinct errors are returned for nonexistent drives, blocks out of range, and hard I/O errors. The *dkblock* and *dkunit* macros once used to support disk interleaving were removed, as interleaving makes no sense with the current file system organization. Messages for recoverable errors, such as soft ECC's, are now handled by *log* instead of *printf*.

### 10.2. General changes in tape drivers

The open functions in the tape drivers now return sensible errors if a drive is in use. They save a pointer to the user's terminal when opened, so that error messages from interrupt level may be printed on the user's terminal using *tprintf*.

### 10.3. Modifications to individual MASSBUS device drivers

- hp.c**      Error recovery in the MASSBUS disk driver is considerably better now than it was. The driver deals with multiple errors in the same transfer much more gracefully. Earlier versions could go into an endless loop correcting one error, then retrying the transfer from the beginning when a second error was encountered. The driver now restarts with the first sector not yet successfully transferred. ECC correction is now possible on bad-sector replacements. The correct sector number is now printed in most error messages. The code to decide whether to initiate a data transfer or whether to do a search was corrected, and the *sdist/rdist* parameters were split into three parameters for each drive: the minimum and maximum rotational distances from the desired sector between which to start a transfer, and the number of sectors to allow after a search before the desired sector. The values chosen for these parameters are probably still not optimal.

There were races when doing a retry on one drive that continued with a repositioning command (recal or seek) and when then beginning a data transfer on another drive. These were corrected by using a distinguished return value, *MBD\_REPOSITION*, from *hpdint* to change the controller state when reverting to positioning operations during a recovery. The remaining steps in the recovery are then managed by *hpus-tart*. Offset commands were previously done under interrupt control, but only on the same retries as recals (every eighth retry starting with the fourth). They are now done on each read retry after the 16th and are done by busy-waiting to avoid the race described above. The tests in the error decoding section of the interrupt handler were rearranged for clarity and to simplify the tests for special conditions such as format operations. The *hpdint* times out if the drive does not become ready after an interrupt rather than hanging at high priority. When forwarding bad sectors, *hpecc*



correctly handles partial-sector transfers; prior versions would transfer a full sector, then continue with a negative byte count, encountering an invalid map register immediately thereafter. Partial-sector transfers are requested by the virtual memory system when swapping page tables.

- mba.c** The top level MASSBUS driver supports the new return code from data-transfer interrupts that indicate a return to positioning commands before restarting a data transfer. It is capable of restarting a transfer after partial completion and adjusting the starting address and byte count according to the amount remaining. It has also been modified to support data transfers in reverse, required for proper error recovery on the TU78. *Mbustart* does not check drives to see that they are present, as dual-ported disks may appear to have a type of zero if the other port is using the disk; in this case, the disk unit start will return MBU\_BUSY.
- mt.c** The TU78 driver has been extensively modified and tested to do better error recovery and to support additional operations.

## 11. VAX UNIBUS device drivers

This section includes changes in device drivers for UNIBUS peripherals other than network interfaces. Modifications common to all of the disk and tape drivers are listed in the previous section on MASSBUS drivers. Many of the UNIBUS drivers were missing null terminations on their lists of standard addresses; this has been corrected.

### 11.1. Changes in terminal multiplexor handling

There are numerous changes that were made uniformly in each of the drivers for UNIBUS terminal multiplexors (DH11, DHU11, DMF32, DMZ32, DZ11 and DZ32). The DMA terminal boards on the same UNIBUS share map registers to map the *clists* to UNIBUS address space. The initialization of *ttys* at open and changes from *ioctls* have been made uniform; the default speed is 9600 baud. Hardware parameters are changed when local modes change; these include LLITOUT and the new LPASS8 options for 8-bit output and input respectively. The code conditional on PORTSELECTOR to accept characters while or before carrier is recognized is the same in all drivers. The processing done for carrier transitions was line discipline-specific, and has been moved into the standard *tty* code; it is called through the previously-unused *l\_modem* entry to the line discipline. This routine's return is used to decide whether to drop DTR. DTR is asserted on lines regardless of the state of the software carrier flag. The drivers for hardware without silo timeouts (DH11, DZ11) dynamically switch between use of the silo during periods of high input and per-character interrupts when input is slow. The timer routines schedule themselves via timeouts and are no longer called directly from the *softclock* interrupt. The timeout runs once per second unless silos are enabled. Hardware faults such as nonexistent memory errors and silo overflows use *log* instead of *printf* to avoid blocking the system at interrupt level.

### 11.2. Changes in individual drivers

- dmf.c** The use of the parallel printer port on the DMF32 is now supported. Autoconfiguration of the DMF includes a test for the sections of the DMF that are present; if only the asynchronous serial ports or parallel printer ports are present, the number of interrupt vectors used is reduced to the minimum number. The common code for the DMF and DMZ drivers was moved to *dmfdmz.c*. Output is done by DMA. The Emulex DMF emulator should work with this driver, despite the incorrect update of the bus address register with odd byte counts. Flow control should work properly with DMA or silo output.
- dmfdmz.c** This file contains common code for the DMF and DMZ drivers.
- dmz.c** This is a new device driver for the DMZ32 terminal multiplexor.
- idc.c** The ECC code for the Integral Disk Controller on the VAX 11/730 was corrected.

kgclock.c	The profiling clock using a DL11 serial interface can be disabled by patching a global variable in the load image before booting or in memory while running. It may thus be used for a profiling run and then turned off. The <i>probe</i> routine returns the correct value now.
lp.c	A fix was made so that slow printers complete printing after device close. The <i>spl</i> 's were cleaned up.
ps.c	The handler for the E & S Picture System 2 has substantial changes to fix refresh problems and clean up the code.
rk.c	Missing entries in the RK07 size table were added.
rl.c	A missing partition was added to the RL02 driver. Drives that aren't spun up during autoconfiguration are now discovered.
rx.c	It is no longer possible to leave a floppy drive locked if no floppy is present at open. Incorrect open counts were corrected.
tm.c	Hacks were added for density selection on Aviv triple-density controllers.
tmscp.c	This is a new driver for tape controllers using the Tape Mass Storage Control Protocol such as the TU81.
ts.c	Adjustment for odd byte addresses when using a buffered data path was incorrect and has been fixed.
uba.c	The UBA_NEED16 flag is tested, and unusable map registers are not allocated for 16-bit addressing devices. Optimizations were made to improve code generation in <i>ubasetup</i> . Zero-vector interrupts on the DW780 now cause resets only when they occur at an unacceptably high rate; this is appreciated by the users who happen to be on the dialups at the time of the 250000th passive release since boot time. UNIBUS memory is now configured separately from devices during autoconfiguration by <i>ubameminit</i> , and this process is repeated after a UNIBUS reset. Devices that consist of UNIBUS memory only may be configured more easily. On a DW780, any map registers made useless by UNIBUS memory above or near them are discarded.
ubareg.h	Definitions were added to include the VAX8600.
ubavar.h	Modifications to the <i>uba_hd</i> structure allow zero vectors and UNIBUS memory allocation to be handled more sensibly. The <i>uba_driver</i> has a new entry for configuration of UNIBUS memory. This routine may probe for UNIBUS memory, and if no further configuration is required may signify the completion of device configuration. A macro was added to extract the UNIBUS address from the value returned by <i>ubasetup</i> and <i>uballoc</i> .
uda.c	This driver is considerably more robust than the one released with 4.2BSD. It configures the drive types so that each type may use its own partition tables. The partitions in the tables as distributed are much more useful, but are mostly incompatible with the previously released driver; a configuration option, RACOMPAT, provides a combination of new and old filesystems for use during conversion. The buffered-data-path handling has been fixed. A dump routine was added.
up.c	Entries were added for the Fujitsu Eagle (2351) in 48-sector mode on an Emulex SC31 controller.
vs.c	This is a driver for the VS100 display on the UNIBUS.

## 12. Bootstrap and standalone utilities

The standalone routines in */sys/stand* and */sys/mdec* have received some work. The bootstrap code is now capable of booting from drives other than drive 0. The device type passed from level to level during the bootstrap operation now encodes the device type, partition number, unit number, and MASSBUS or UNIBUS adaptor number (one byte for each field, from least significant to most significant). The bootstrap is much faster, as the standalone *read* operation uses raw I/O when



possible.

The formatter has been much improved. It deals with skip-sector devices correctly; the previous version tested for skip-sector capability incorrectly, and thus never dealt with it. The formatter is capable of formatting sections of the disk, track by track, and can run a variable number of passes. The error retry logic in the standalone disk drivers was corrected and parameterized so that the formatter may disable most corrections.



## A Fast File System for UNIX\*

Marshall Kirk McKusick, William N. Joy†,  
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

### ABSTRACT

A reimplementa-tion of the UNIX file system is described. The reimplementa-tion provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the programmers' interface are discussed. These include a mechanism to place advisory locks on files, extensions of the name space across file systems, the ability to use long file names, and provisions for administrative control of resource usage.

Revised February 18, 1984

CR Categories and Subject Descriptors: D.4.3 [Operating Systems]: File Systems Management – *file organization, directory structures, access methods*; D.4.2 [Operating Systems]: Storage Management – *allocation/deallocation strategies, secondary storage devices*; D.4.8 [Operating Systems]: Performance – *measurements, operational analysis*; H.3.2 [Information Systems]: Information Storage – *file organization*

Additional Keywords and Phrases: UNIX, file system organization, file system performance, file system design, application program interface.

General Terms: file system, measurement, performance.

\* UNIX is a trademark of Bell Laboratories.

† William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡ Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.



## TABLE OF CONTENTS

1. Introduction
2. Old file system
3. New file system organization
  - 3.1. Optimizing storage utilization
  - 3.2. File system parameterization
  - 3.3. Layout policies
4. Performance
5. File system functional enhancements
  - 5.1. Long file names
  - 5.2. File locking
  - 5.3. Symbolic links
  - 5.4. Rename
  - 5.5. Quotas

## References

## 1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to effect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the facilities that are available to programmers.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. Virtually no constraints other than available disk space are placed on file growth [Ritchie74], [Thompson78].\*

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications such as VLSI design and image processing do a small amount of processing on a large quantities of data and need to have a high throughput from the file system. High throughput rates are also needed by programs that map files from the file system into large virtual address spaces. Paging data in and out of the file system is likely to occur frequently [Ferrin82b]. This requires a file system providing higher bandwidth than the original 512 byte UNIX one that provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently, users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. Previous work to improve the UNIX file system performance has been done by [Ferrin82a]. The UNIX operating system drew many of its ideas from Multics, a large, high

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

\* In practice, a file's size is constrained to be less than about one gigabyte.



performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a LISP environment [Symbolics81]. A good introduction to the physical latencies of disks is described in [Pechura83].

## 2. Old File System

In the file system developed at Bell Laboratories (the "traditional" file system), each disk drive is divided into one or more partitions. Each of these disk partitions may contain one file system. A file system never spans multiple partitions.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to the *free list*, a linked list of all the free blocks in the file system.

Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. An inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in an inode itself\*. An inode may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further singly indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A 150 megabyte traditional UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from the file's inode to its data. Files in a single directory are not typically allocated consecutive slots in the 4 megabytes of inodes, causing many non-consecutive blocks of inodes to be accessed when executing operations on the inodes of several files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by staging modifications to critical file system information so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors: each disk transfer accessed twice as much data, and most files could be described without need to access indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random, causing files to have their blocks allocated randomly over the disk. This forced a seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate

† By "partition" here we refer to the subdivision of physical space on a disk drive. In the traditional file system, as in the new file system, file systems are really located in logical disk partitions that may overlap. This overlapping is made available, for example, to allow programs to copy entire disk drives containing multiple file systems.

\* The actual number may vary from system to system, but is usually in the range 5-13.



deteriorated to 30 kilobytes per second after a few weeks of moderate use because of this randomization of data block placement. There was no way of restoring the performance of an old file system except to dump, rebuild, and restore the file system. Another possibility, as suggested by [Maruyama76], would be to have a process that periodically reorganized the data on the disk to restore locality.

### 3. New file system organization

In the new file system organization (as in the old file system organization), each disk drive contains one or more file systems. A file system is described by its super-block, located at the beginning of the file system's disk partition. Because the super-block contains critical data, it is replicated to protect against catastrophic loss. This is done when the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as  $2^{32}$  bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of a file system is recorded in the file system's super-block so it is possible for file systems with different block sizes to be simultaneously accessible on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization divides a disk partition into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. The bit map of available blocks in the cylinder group replaces the traditional file system's free list. For each cylinder group a static number of inodes is allocated at file system creation time. The default policy is to allocate one inode for each 2048 bytes of space in the cylinder group, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all redundant copies of the super-block. Thus the cylinder group bookkeeping information begins at a varying offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group than the preceding cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-block. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

#### 3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transaction, greatly increasing file system throughput. As an example, consider a file in the new file system composed of

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16 kilobytes or greater. This is because of a requirement that the first 8 kilobytes of the disk be reserved for a bootstrap program and a separate requirement that the cylinder group information begin on a file system block boundary. To start the cylinder group on a file system block boundary, file systems with block sizes larger than 8 kilobytes would have to leave an empty space between the end of the boot block and the beginning of the cylinder group. Without knowing the size of the file system blocks, the system would not know what roundup function to use to find the beginning of the first cylinder group.

4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before requiring a seek.

The main problem with larger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The files measured to obtain these figures reside on one of our time sharing systems that has roughly 1.2 gigabytes of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formatted space.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	Data + inodes, 512 byte block UNIX file system
866.5 Mb	11.8	Data + inodes, 1024 byte block UNIX file system
948.5 Mb	22.4	Data + inodes, 2048 byte block UNIX file system
1128.3 Mb	45.6	Data + inodes, 4096 byte block UNIX file system

Table 1 – Amount of wasted space as a function of block size.

The space wasted is calculated to be the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can optionally be broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space available in a cylinder group at the fragment level; to determine if a block is available, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0-5, 10, and 11 are in use, while fragments 6-9, and 12-15 are free. Fragments of adjoining blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6-9 cannot be allocated as a full block; only fragments 12-15 can be coalesced into a full block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remaining fragments of the block are made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would use two full size blocks and one three fragment portion of another block. If no block with three aligned fragments is available at the time the file is created, a full size block is split yielding the necessary fragments and a single unused fragment. This remaining fragment can be allocated to another file as needed.

Space is allocated to a file when a program does a *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased\*. If the file needs to be expanded to

\* A program may be overwriting data in the middle of an existing file in which case space would already have been allocated.



hold the new data, one of three conditions exists:

- 1) There is enough space left in an already allocated block or fragment to hold the new data. The new data is written into the available space.
- 2) The file contains no fragmented blocks (and the last block in the file contains insufficient space to hold the new data). If space exists in a block already allocated, the space is filled with new data. If the remainder of the new data contains more than a full block of data, a full block is allocated and the first full block of new data is written there. This process is repeated until less than a full block of new data remains. If the remaining new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The remaining new data is written into the located space.
- 3) The file contains one or more fragments (and the fragments contain insufficient space to hold the new data). If the size of the new data plus the size of the data already in the fragments exceeds the size of a full block, a new block is allocated. The contents of the fragments are copied to the beginning of the block and the remainder of the block is filled with new data. The process then continues as in (2) above. Otherwise, if the new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The contents of the existing fragments appended with the new data are written into the allocated space.

The problem with expanding a file one fragment at a time is that data may be copied many times as a fragmented block expands to a full block. Fragment reallocation can be minimized if the user program writes a full block at a time, except for a partial block at the end of the file. Since file systems with different block sizes may reside on the same system, the file system interface has been extended to provide application programs the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The amount of wasted space in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of wasted space as the 512 byte block UNIX file system. The new file system uses less space than the 512 byte or 1024 byte file systems for indexing information for large files and the same amount of space for small files. These savings are offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when a new file system's fragment size equals an old file system's block size.

In order for the layout policies to be effective, a file system cannot be kept completely full. For each file system there is a parameter, termed the free space reserve, that gives the minimum acceptable percentage of file system blocks that should be free. If the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter may be changed at any time, even when the file system is mounted and active. The transfer rates that appear in section 4 were measured on file systems kept less than 90% full (a reserve of 10%). If the number of free blocks falls to zero, the file system throughput tends to be cut in half, because of the inability of the file system to localize blocks in a file. If a file system's performance degrades because of overfilling, it may be restored by removing files until the amount of free space once again reaches the minimum acceptable level. Access rates for files created during periods of little free space may be restored by moving their data once enough space is available. The free space reserve must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, the percentage of waste in an old 1024 byte UNIX file system is roughly comparable to a new 4096/512 byte file system with the free space reserve set at 5%. (Compare 11.8% wasted with the old file system to 6.9% waste + 5% reserved space in the new file system.)

### 3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration-dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can be adapted to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be rotationally well positioned. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with an input/output channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks can often be accessed without suffering lost time because of an intervening disk revolution. For processors without input/output channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to service an interrupt and schedule a new disk transfer. Given a block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in the file will come into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the available blocks in a cylinder group at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive. The super-block contains a vector of lists called *rotational layout tables*. The vector is indexed by rotational position. Each component of the vector lists the index into the block map for every data block contained in its rotational position. When looking for an allocatable block, the system first looks through the summary counts for a rotational position with a non-zero block count. It then uses the index of the rotational position to find the appropriate list to use to index through only the relevant parts of the block map to find a free block.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with a rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

### 3.3. Layout policies

The file system layout policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a



long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalaincn77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Inodes of files in the same directory are frequently accessed together. For example, the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the inodes of files in a directory in the same cylinder group. To ensure that files are distributed throughout the disk, a different policy is used for directory allocation. A new directory is placed in a cylinder group that has a greater than average number of free inodes, and the smallest number of directories already in it. The intent of this policy is to allow the inode clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for a particular cylinder group can be read with 8 to 16 disk transfers. (At most 16 disk transfers are required because a cylinder group may have no more than 2048 inodes.) This puts a small and constant upper bound on the number of disk transfers required to access the inodes for all the files in a directory. In contrast, the old file system typically requires one disk transfer to fetch the inode for each file in a directory.

The other major resource is data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Further, using all the space in a cylinder group causes future allocations for any file in the cylinder group to also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The heuristic solution chosen is to redirect block allocation to a different cylinder group when a file exceeds 48 kilobytes, and at every megabyte thereafter.\* The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free, otherwise it allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristics that employ only partial information.

If a requested block is not available, the local allocator uses a four level allocation strategy:

\* The first spill over point at 48 kilobytes is the point at which a file on a 4096 byte block file system first requires a single indirect block. This appears to be a natural first point at which to redirect block allocation. The other spillover points are chosen with the intent of forcing block allocation to be redirected when a file has used about 25% of the data blocks in a cylinder group. In observing the new file system in day to day use, the heuristics appear to work well in minimizing the number of completely filled cylinder groups.

- 1) Use the next available block rotationally closest to the requested block on the same cylinder. It is assumed here that head switching time is zero. On disk controllers where this is not the case, it may be possible to incorporate the time required to switch between disk platters when constructing the rotational layout tables. This, however, has not yet been tried.
- 2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
- 3) If that cylinder group is entirely full, quadratically hash the cylinder group number to choose another cylinder group to look for a free block.
- 4) Finally if the hash fails, apply an exhaustive search to all cylinder groups.

Quadratic hash is used because of its speed in finding unused slots in nearly full hash tables [Knuth75]. File systems that are parameterized to maintain at least 10% free space rarely use this strategy. File systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random; the most important characteristic of the strategy used under such conditions is that the strategy be fast.

#### 4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empirical studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories (to force the system to access inodes in multiple cylinder groups), the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate at which user programs can transfer data to or from a file without performing any processing on it. These programs must read and write enough data to insure that buffering in the operating system does not affect the results. They are also run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The tests used and their results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the CPU or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an AMPEX Capricorn 330 megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured. The same number of system calls were performed in all tests; the basic system call overhead was a negligible portion of the total running time of the tests.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system with a 10% free space reserve. Synthetic work loads suggest that throughput deteriorates to about half the rates given in Table 2 when the file systems are full.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is calculated by multiplying the number of bytes on a track by the number of revolutions of the disk per second. The

† A UNIX command that is similar to the reading test that we used is "cp file /dev/null", where "file" is eight megabytes long.



Type of File System	Processor and Bus Measured	Speed	Read Bandwidth	% CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/983 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/983 22%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/983 24%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Speed	Write Bandwidth	% CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/983 5%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/983 14%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/983 22%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/983 33%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3–5% of the disk bandwidth, while the new file system uses up to 47% of the bandwidth.

Both reads and writes are faster in the new system than in the old system. The biggest factor in this speedup is because of the larger block size used by the new file system. The overhead of allocating blocks in the new system is greater than the overhead of allocating blocks in the old system, however fewer blocks need to be allocated in the new system because they are bigger. The net effect is that the cost per byte allocated is about the same for both systems.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, making the processor unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the write system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers queue up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek distance, the average seek between the scheduled disk writes is much less than it would be if the data blocks were written out in the random disk order in which they are generated. However when the file is read, the read system call is processed synchronously so the disk blocks must be retrieved from the disk in the non-optimal seek order in which they are requested. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

In the new system the blocks of a file are more optimally ordered on the disk. Even though reads are still synchronous, the requests are presented to the disk in a much better order. Even though the writes are still asynchronous, they are already presented to the disk in minimum seek order so there is no gain to be had by reordering them. Hence the disk seek latencies that limited the old file system have little effect in the new file system. The cost of allocation is the factor in the new system that causes writes to be slower than reads.

The performance of the new file system is currently limited by memory to memory copy operations required to move data from disk buffers in the system's address space to data buffers in the user's address space. These copy operations account for about 40% of the time spent performing an input/output operation. If the buffers in both address spaces were properly aligned, this transfer could be performed without copying by using the VAX virtual memory management hardware. This



would be especially desirable when transferring large amounts of data. We did not implement this because it would change the user interface to the file system in two major ways: user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow contiguous disk blocks to be read in a single disk transaction. Many disks used with UNIX systems contain either 32 or 48 512 byte sectors per track. Each track holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than 50% of the available bandwidth. If the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up allocations, the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79]. This technique was not included because block allocation currently accounts for less than 10% of the time spent in a write system call and, once again, the current throughput rates are already limited by the speed of the available processors.

## 5. File system functional enhancements

The performance enhancements to the UNIX file system did not require any changes to the semantics or data structures visible to application programs. However, several changes had been generally desired for some time but had not been introduced because they would require users to dump and restore all their file systems. Since the new file system already required all existing file systems to be dumped and restored, these functional enhancements were introduced at this time.

### 5.1. Long file names

File names can now be of nearly arbitrary length. Only programs that read directories are affected by this change. To promote portability to UNIX systems that are not running the new file system, a set of directory access routines have been introduced to provide a consistent interface to directories on both old and new systems.

Directories are allocated in 512 byte units called chunks. This size is chosen so that each allocation can be transferred to disk in a single operation. Chunks are broken up into variable length records termed directory entries. A directory entry contains the information necessary to map the name of a file to its associated inode. No directory entry is allowed to span multiple chunks. The first three fields of a directory entry are fixed length and contain: an inode number, the size of the entry, and the length of the file name contained in the entry. The remainder of an entry is variable length and contains a null terminated file name, padded to a 4 byte boundary. The maximum length of a file name in a directory is currently 255 characters.

Available space in a directory is recorded by having one or more entries accumulate the free space in their entry size fields. This results in directory entries that are larger than required to hold the entry name plus fixed length fields. Space allocated to a directory should always be completely accounted for by totaling up the sizes of its entries. When an entry is deleted from a directory, its space is returned to a previous entry in the same directory chunk by increasing the size of the



previous entry by the size of the deleted entry. If the first entry of a directory chunk is free, then the entry's inode number is set to zero to indicate that it is unallocated.

## 5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to use a separate "lock" file. A process would try to create a "lock" file. If the creation succeeded, then the process could proceed with its update; if the creation failed, then the process would wait and try again. This mechanism had three drawbacks. Processes consumed CPU time by looping over attempts to create locks. Locks left lying around because of system crashes had to be manually removed (normally in a system startup command script). Finally, processes running as system administrator are always permitted to create files, so were forced to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight forward, so a mechanism for locking files has been added.

The most general schemes allow multiple processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to serialize access to a file with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the standard system applications, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the extent of enforcement. A hard lock is always enforced when a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel. With advisory locks the policy is left to the user programs. In the UNIX system, programs with system administrator privilege are allowed override any protection scheme. Because many of the programs that need to use locks must also run as the system administrator, we chose to implement advisory locks rather than create an additional protection scheme that was inconsistent with the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process may have an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the lock request will block until the lock can be obtained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process may access the file.

Locks are applied or removed only on open files. This means that locks can be manipulated without needing to close and reopen a file. This is useful, for example, when a process wishes to apply a shared lock, read some information and determine whether an update is required, then apply an exclusive lock and update the file.

A request for a lock will cause a process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to conditionally request a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since locks exist only while the locking processes exist, lock files can never be left active after the processes exit or if the system crashes.

Almost no deadlock detection is attempted. The only deadlock detection done by the system is that the file to which a lock is applied must not already have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail).

### 5.3. Symbolic links

The traditional UNIX file system allows multiple directory entries in the same file system to reference a single file. Each directory entry “links” a file’s name to an inode and its contents. The link concept is fundamental; inodes do not reside in directories, but exist separately and are referenced by links. When all the links to an inode are removed, the inode is deallocated. This style of referencing an inode does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* similar to the scheme used by Multics [Feiertag71] have been added.

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. In UNIX, pathnames are specified relative to the root of the file system hierarchy, or relative to a process’s current working directory. Pathnames specified relative to the root are called absolute pathnames. Pathnames specified relative to the current working directory are termed relative pathnames. If a symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links; seven system utilities required changes to use these calls.

In future Berkeley software distributions it may be possible to reference file systems located on remote machines using pathnames. When this occurs, it will be possible to create symbolic links that span machines.

### 5.4. Rename

Programs that create a new version of an existing file typically create the new version as a temporary file and then rename the temporary file with the name of the target file. In the old UNIX file system renaming required three calls to the system. If a program were interrupted or the system crashed between these calls, the target file could be left with only its temporary name. To eliminate this possibility the *rename* system call has been added. The rename call does the rename operation in a fashion that guarantees the existence of the target name.

Rename works both on data files and directories. When renaming directories, the system must do special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the descendants of the target directory to insure that it does not include the directory being moved.

### 5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of inodes and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Resources are given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more resources while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If users fails to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

## Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when file systems were less stable than they should have been. The criticisms and suggestions by the reviews contributed significantly to the coherence of the paper. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

## References

- [Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.
- [Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.
- [Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41
- [Ferrin82a] Ferrin, T.E., "Performance and Robustness Improvements in Version 7 UNIX", Computer Graphics Laboratory Technical Report 2, School of Pharmacy, University of California, San Francisco, January 1982. Presented at the 1982 Winter Usenix Conference, Santa Monica, California.
- [Ferrin82b] Ferrin, T.E., "Performance Issues of VMUNIX Revisited", ;login: (The Usenix Association Newsletter), Vol 7, #5, November 1982. pp 3-6
- [Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.
- [Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978
- [Knuth75] Kunth, D. "The Art of Computer Programming", Volume 3 - Sorting and Searching, Addison-Wesley Publishing Company Inc, Reading, Mass, 1975. pp 506-549
- [Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", CACM, 19, 11. Nov 1976. pp 634-642
- [Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247
- [Pechura83] Pechura, M., and Schoeffler, J. "Estimating File Access Time of Floppy Disks", CACM, 26, 10. Oct 1983. pp 754-763
- [Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55
- [Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42
- [Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

- [Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117
- [Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54
- [Symbolics81] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.
- [Thompson78] Thompson, K. "UNIX Implementation", Bell System Technical Journal, 57, 6, part 2. pp 1931-1946 July-August 1978.
- [Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Department of Computer Science, Pittsburg, PA 15213 #CMU-CS-80, Sept 1980.
- [Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473
- [White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.





## Networking Implementation Notes 4.3BSD Edition

*Samuel J. Leffler, William N. Joy, Robert S. Fabry, and Michael J. Karels*

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

### ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.3BSD version of the UNIX\* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The "Berkeley Software Architecture Manual, 4.3BSD Edition" (PS1:6) provides a description of the user interface to the networking facilities.

Revised June 5, 1986

---

\* UNIX is a trademark of Bell Laboratories.

† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.



**TABLE OF CONTENTS**

- 1. Introduction**
- 2. Overview**
- 3. Goals**
- 4. Internal address representation**
- 5. Memory management**
- 6. Internal layering**
  - 6.1. Socket layer
    - 6.1.1. Socket state
    - 6.1.2. Socket data queues
    - 6.1.3. Socket connection queuing
  - 6.2. Protocol layer(s)
  - 6.3. Network-interface layer
    - 6.3.1. UNIBUS interfaces
- 7. Socket/protocol interface**
- 8. Protocol/protocol interface**
  - 8.1. pr\_output
  - 8.2. pr\_input
  - 8.3. pr\_ctlinput
  - 8.4. pr\_ctloutput
- 9. Protocol/network-interface interface**
  - 9.1. Packet transmission
  - 9.2. Packet reception
- 10. Gateways and routing issues**
  - 10.1. Routing tables
  - 10.2. Routing table interface
  - 10.3. User level routing policies
- 11. Raw sockets**
  - 11.1. Control blocks
  - 11.2. Input processing
  - 11.3. Output processing
- 12. Buffering and congestion control**
  - 12.1. Memory management
  - 12.2. Protocol buffering policies
  - 12.3. Queue limiting
  - 12.4. Packet forwarding
- 13. Out of band data**
- 14. Trailer protocols**
- Acknowledgements**
- References**



## 1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX, as modified in the 4.3BSD release. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *Berkeley Software Architecture Manual, 4.3BSD Edition* [Joy86]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

## 2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

## 3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

#### 4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
    short    sa_family;    /* data format identifier */
    char     sa_data[14];  /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa\_family* field indicates the address family to which the address belongs, and the *sa\_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats.\* Specific address formats use private structure definitions that define the format of the data field. The system interface supports larger address structures, although address-family-independent support facilities, for example routing and raw socket interfaces, provide only 14 bytes for address storage. Protocols that do not use those facilities (e.g, the current Unix domain) may use larger data areas.

#### 5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
    struct    mbuf *m_next;    /* next buffer in chain */
    u_long    m_off;           /* offset of data */
    short     m_len;           /* amount of data in this mbuf */
    short     m_type;          /* mbuf type (accounting) */
    u_char    m_dat[MLEN];     /* data storage */
    struct    mbuf *m_act;     /* link in higher-level mbuf list */
};
```

The *m\_next* field is used to chain mbufs together on linked lists, while the *m\_act* field allows lists of mbuf chains to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m\_next* field, while groups of objects are linked via the *m\_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m\_dat*. The *m\_len* field indicates the amount of data, while the *m\_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define mtod(x,t) ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, which is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. An mbuf with an external data area may be recognized by the larger offset to the data area; this is formalized by the macro *M\_HASCL(m)*, which is true if the mbuf whose address is *m* has an external page cluster. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the reference to the data and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate

\* Later versions of the system may support variable length addresses.

mbufs are not normally aware whether data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following may be used to allocate and free mbufs:

```
m = m_get(wait, type);
MGET(m, wait, type);
```

The subroutine *m\_get* and the macro *MGET* each allocate an mbuf, placing its address in *m*. The argument *wait* is either *M\_WAIT* or *M\_DONTWAIT* according to whether allocation should block or fail if no mbuf is available. The *type* is one of the predefined mbuf types for use in accounting of mbuf allocation.

```
MCLGET(m);
```

This macro attempts to allocate an mbuf page cluster to associate with the mbuf *m*. If successful, the length of the mbuf is set to *CLSIZE*, the size of the page cluster.

```
n = m_free(m);
MFREE(m,n);
```

The routine *m\_free* and the macro *MFREE* each free a single mbuf, *m*, and any associated external storage area, placing a pointer to its successor in the chain it heads, if any, in *n*.

```
m_freem(m);
```

This routine frees an mbuf chain headed by *m*.

The following utility routines are available for manipulating mbuf chains:

```
m = m_copy(m0, off, len);
```

The *m\_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as *M\_COPYALL*, all the data present, offset as before, is copied.

```
m_cat(m, n);
```

The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

```
m_adj(m, diff);
```

The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation; alterations are accomplished by changing the *m\_len* and *m\_off* fields of mbufs.

```
m = m_pullup(m0, size);
```

After a successful call to *m\_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory within the data area of the mbuf (allowing access via a pointer, obtained using the *mtod* macro, and allowing the mbuf to be located from a pointer to the data area using *dtom*, defined below). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m\_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring that mbufs always reside on 128 byte boundaries, it is always possible to locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. Note that this works only with objects stored in the internal data buffer of the mbuf. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define dtom(x) ((struct mbuf*)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets as well as memory allocated for packets and headers. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat(1)* program.

## 6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces to which each must conform.

### 6.1. Socket layer

The socket layer deals with the interprocess communication facilities provided by the system. A socket is a bidirectional endpoint of communication which is “typed” by the semantics of communication it supports. The system calls described in the *Berkeley Software Architecture Manual* [Joy86] are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
    short    so_type;           /* generic type */
    short    so_options;        /* from socket call */
    short    so_linger;         /* time to linger while closing */
    short    so_state;          /* internal state flags */
    caddr_t  so_pcb;            /* protocol control block */
    struct    protosw *so_proto; /* protocol handle */
    struct    socket *so_head;   /* back pointer to accept socket */
    struct    socket *so_q0;     /* queue of partial connections */
    short    so_q0len;          /* partials on so_q0 */
    struct    socket *so_q;      /* queue of incoming connections */
    short    so_qlen;           /* number of connections on so_q */
    short    so_qlimit;         /* max number queued connections */
    struct    sockbuf so_rcv;     /* receive queue */
    struct    sockbuf so_snd;    /* send queue */
    short    so_timeo;          /* connection timeout */
    u_short  so_error;          /* error affecting connection */
    u_short  so_oobmark;        /* chars to oob mark */
    short    so_pgrp;           /* pgrp for signals */
};
```

Each socket contains two data queues, *so\_rcv* and *so\_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so\_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol-specific data structure, the “protocol control block,” is also present in the socket structure. Protocols control this data structure, which normally includes a back pointer to the parent socket structure to allow easy lookup when returning information to a user (for example, placing an error number in the *so\_error* field). The other entries in the socket structure are used in queuing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket’s state.

Processes “rendezvous at a socket” in many instances. For instance, when a process wishes to extract data from a socket’s receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as a “wait channel” to be used in notification. When data arrives for the process and is placed in the socket’s queue, the blocked process is identified by the fact it is waiting “on the queue.”

#### 6.1.1. Socket state

A socket’s state is defined from the following:

```
#define SS_NOFDREF      0x001 /* no file table ref any more */
#define SS_ISCONNECTED  0x002 /* socket connected to a peer */
#define SS_ISCONNECTING 0x004 /* in process of connecting to peer */
#define SS_ISDISCONNECTING 0x008 /* in process of disconnecting */
#define SS_CANTSENDMORE 0x010 /* can't send more data to peer */
#define SS_CANTRCVMORE  0x020 /* can't receive more data from peer */
#define SS_RCVATMARK    0x040 /* at mark on input */

#define SS_PRIV          0x080 /* privileged */
#define SS_NBIO          0x100 /* non-blocking ops */
#define SS_ASYNC         0x200 /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created, the state is defined based on the type of socket. It may change as control actions are performed, for example connection establishment. It may also change according to the type of input/output the user wishes to perform, as indicated by options set with *fcntl*. “Non-blocking” I/O implies that a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error *EWOULDBLOCK*, or the service request may be partially fulfilled, e.g. a request for more data than is present.

If a process requested “asynchronous” notification of events related to the socket, the *SIGIO* signal is posted to the process when such events occur. An event is a change in the socket’s state; examples of such occurrences are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked “privileged” if it was created by the super-user. Only privileged sockets may bind addresses in privileged portions of an address space or use “raw” sockets to access lower levels of the network.

#### 6.1.2. Socket data queues

A socket’s data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
    u_short    sb_cc;           /* actual chars in buffer */
    u_short    sb_hiwat;       /* max actual char count */
    u_short    sb_mbcnt;       /* chars of mbufs used */
    u_short    sb_mbmax;       /* max chars of mbufs to use */
    u_short    sb_lowat;       /* low water mark */
    short      sb_timeo;       /* timeout */
    struct      mbuf *sb_mb;    /* the mbuf chain */
    struct      proc *sb_sel;   /* process selecting read/write */
    short      sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of data characters as well as high and low water marks are used by the protocols in controlling the flow of data. The amount of buffer space (characters of mbufs and associated data pages) is also recorded along with the limit on

buffer allocation. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).\*

When a socket is created, the supporting protocol “reserves” space for the send and receive queues of the socket. The limit on buffer allocation is set somewhat higher than the limit on data characters to account for the granularity of buffer allocation. The actual storage associated with a socket queue may fluctuate during a socket’s lifetime, but it is assumed that this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

Data queued at a socket is stored in one of two styles. Stream-oriented sockets queue data with no addresses, headers or record boundaries. The data are in mbufs linked through the *m\_next* field. Buffers containing access rights may be present within the chain if the underlying protocol supports passage of access rights. Record-oriented sockets, including datagram sockets, queue data as a list of packets; the sections of packets are distinguished by the types of the mbufs containing them. The mbufs which comprise a record are linked through the *m\_next* field; records are linked from the *m\_act* field of the first mbuf of one packet to the first mbuf of the next. Each packet begins with an mbuf containing the “from” address if the protocol provides it, then any buffers containing access rights, and finally any buffers containing data. If a record contains no data, no data buffers are required unless neither address nor access rights are present.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources:

```
#define SB_LOCK          0x01 /* lock on data queue (so_rcv only) */
#define SB_WANT          0x02 /* someone is waiting to lock */
#define SB_WAIT          0x04 /* someone is waiting for data/space */
#define SB_SEL           0x08 /* buffer is selected */
#define SB_COLL          0x10 /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

### 6.1.3. Socket connection queuing

In dealing with connection oriented sockets (e.g. SOCK\_STREAM) the two ends are considered distinct. One end is termed *active*, and generates connection requests. The other end is called *passive* and accepts connection requests.

From the passive side, a socket is marked with SO\_ACCEPTCONN when a *listen* call is made, creating two queues of sockets: *so\_q0* for connections in progress and *so\_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so\_q0* by calling the routine *sonewconn()*. When the connection is established, the socket structure is then transferred to *so\_q*, making it available for an *accept*.

If an SO\_ACCEPTCONN socket is closed with sockets on either *so\_q0* or *so\_q*, these sockets are dropped, with notification to the peers as appropriate.

### 6.2. Protocol layer(s)

Each socket is created in a communications domain, which usually implies both an addressing structure (address family) and a set of protocols which implement various socket types within the domain (protocol family). Each domain is defined by the following structure:

\* The low-water mark is always presumed to be 0 in the current implementation.

```

struct domain (
    int    dom_family;        /* PF_XXX */
    char   *dom_name;
    int    (*dom_init)();      /* initialize domain data structures */
    int    (*dom_externalize)(); /* externalize access rights */
    int    (*dom_dispose)();   /* dispose of internalized rights */
    struct protosw *dom_protosw, *dom_protoswnPROTOSW;
    struct domain *dom_next;
);

```

At boot time, each domain configured into the kernel is added to a linked list of domain. The initialization procedure of each domain is then called. After that time, the domain structure is used to locate protocols within the protocol family. It may also contain procedure references for externalization of access rights at the receiving socket and the disposal of access rights that are not received.

Protocols are described by a set of entry points and certain socket-visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```

struct protosw (
    short pr_type;            /* socket type used for */
    struct domain *pr_domain; /* domain protocol a member of */
    short pr_protocol;        /* protocol number */
    short pr_flags;           /* socket visible attributes */
    /* protocol-protocol hooks */
    int    (*pr_input)();      /* input to protocol (from below) */
    int    (*pr_output)();     /* output to protocol (from above) */
    int    (*pr_ctlinput)();   /* control input (from below) */
    int    (*pr_ctloutput)();  /* control output (from above) */
    /* user-protocol hook */
    int    (*pr_usrreq)();     /* user request */
    /* utility hooks */
    int    (*pr_init)();       /* initialization routine */
    int    (*pr_fasttimo)();   /* fast timeout (200ms) */
    int    (*pr_slowtimo)();   /* slow timeout (500ms) */
    int    (*pr_drain)();      /* flush any excess space possible */
);

```

A protocol is called through the *pr\_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr\_fasttimo* entry and every 500 milliseconds through the *pr\_slowtimo* for timer based actions. The system will call the *pr\_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr\_input* and *pr\_output* routines. *Pr\_input* passes data up (towards the user) and *pr\_output* passes it down (towards the network); control information passes up and down on *pr\_ctlinput* and *pr\_ctloutput*. The protocol is responsible for the space occupied by any of the arguments to these entries and must either pass it onward or dispose of it. (On output, the lowest level reached must free buffers storing the arguments; on input, the highest level is responsible for freeing buffers.)

The *pr\_usrreq* routine interfaces protocols to the socket code and is described below.

The *pr\_flags* field is constructed from the following values:

```

#define PR_ATOMIC      0x01    /* exchange atomic messages only */
#define PR_ADDR        0x02    /* addresses given with messages */
#define PR_CONNREQUIRED 0x04    /* connection required by protocol */
#define PR_WANTRCVD    0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS      0x10    /* passes capabilities */

```

Protocols which are connection-based specify the `PR_CONNREQUIRED` flag so that the socket routines will never attempt to send data before a connection has been established. If the `PR_WANTRCVD` flag is set, the socket routines will notify the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The `PR_ADDR` field indicates that any data placed in the socket's receive queue will be preceded by the address of the sender. The `PR_ATOMIC` flag specifies that each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The `PR_RIGHTS` flag indicates that the protocol supports the passing of capabilities; this is currently used only by the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table for the domain looking for an appropriate protocol to support the type of socket being created. The *pr\_type* field contains one of the possible socket types (e.g. `SOCK_STREAM`), while the *pr\_domain* is a back pointer to the domain structure. The *pr\_protocol* field contains the protocol number of the protocol, normally a well-known value.

### 6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and decapsulation of any link-layer header information required to deliver a message to its destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. An interface may have addresses in one or more address families. The address is set at boot time using an *ioctl* on a socket in the appropriate domain; this operation is implemented by the protocol family, after verifying the operation through the device *ioctl* entry.

An interface is defined by the following structure,



```

struct ifnet {
    char    *if_name;           /* name, e.g. "en" or "lo" */
    short   if_unit;            /* sub-unit for lower level driver */
    short   if_mtu;             /* maximum transmission unit */
    short   if_flags;           /* up/down, broadcast, etc. */
    short   if_timer;           /* time 'til if_watchdog called */
    struct  ifaddr *if_addrlist; /* list of addresses of interface */
    struct  ifqueue if_snd;      /* output queue */
    int     (*if_init)();        /* init routine */
    int     (*if_output)();      /* output routine */
    int     (*if_ioctl)();       /* ioctl routine */
    int     (*if_reset)();       /* bus reset routine */
    int     (*if_watchdog)();    /* timer routine */
    int     if_ipackets;         /* packets received on interface */
    int     if_ierrors;         /* input errors on interface */
    int     if_opackets;        /* packets sent on interface */
    int     if_oerrors;         /* output errors on interface */
    int     if_collisions;       /* collisions on csma interfaces */
    struct  ifnet *if_next;
};

```

Each interface address has the following form:

```

struct ifaddr {
    struct  sockaddr ifa_addr; /* address of interface */
    union {
        struct  sockaddr ifu_broadaddr;
        struct  sockaddr ifu_dstaddr;
    } ifa_ifu;
    struct  ifnet *ifa_ifp;     /* back-pointer to interface */
    struct  ifaddr *ifa_next;   /* next address for interface */
};
#define ifa_broadaddr ifa_ifu.ifu_broadaddr /* broadcast address */
#define ifa_dstaddr ifa_ifu.ifu_dstaddr /* other end of p-to-p link */

```

The protocol generally maintains this structure as part of a larger structure containing additional information concerning the address.

Each interface has a send queue and routines used for initialization, *if\_init*, and output, *if\_output*. If the interface resides on a system bus, the routine *if\_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if\_watchdog*; if *if\_timer* is non-zero, it is decremented once per second until it reaches zero, at which time the watchdog routine is called.

The state of an interface and certain characteristics are stored in the *if\_flags* field. The following values are possible:

#define	IFF_UP	0x1	/* interface is up */
#define	IFF_BROADCAST	0x2	/* broadcast is possible */
#define	IFF_DEBUG	0x4	/* turn on debugging */
#define	IFF_LOOPBACK	0x8	/* is a loopback net */
#define	IFF_POINTOPOINT	0x10	/* interface is point-to-point link */
#define	IFF_NOTRAILERS	0x20	/* avoid use of trailers */
#define	IFF_RUNNING	0x40	/* resources allocated */
#define	IFF_NOARP	0x80	/* no address resolution protocol */

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF\_BROADCAST flag will be set and the *ifa\_broadaddr* field will contain the address to be used in

sending or accepting a broadcast packet. If the interface is associated with a point-to-point hardware link (for example, a DEC DMR-11), the `IFF_POINTOPOINT` flag will be set and `ifa_dstaddr` will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, `if_addr`, are used in filtering incoming packets. The interface sets `IFF_RUNNING` after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The `IFF_NOTRAILERS` flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets, or (where per-host negotiation of trailers is possible) that trailer encapsulations should not be requested; *trailer* protocols are described in section 14. The `IFF_NOARP` flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

Various statistics are also stored in the interface structure. These may be viewed by users using the `netstat(1)` program.

The interface address and flags may be set with the `SIOCSIFADDR` and `SIOCSIFFLAGS` *ioctl*s. `SIOCSIFADDR` is used initially to define each interface's address; `SIOCSIFFLAGS` can be used to mark an interface down and perform site-specific configuration. The destination address of a point-to-point link is set with `SIOCSIFDSTADDR`. Corresponding operations exist to read each value. Protocol families may also support operations to set and read the broadcast address. In addition, the `SIOCGIFCONF` *ioctl* retrieves a list of interface names and addresses for all interfaces and protocols on the host.

### 6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct ifubinfo {
    short    iff_uban;           /* uba number */
    short    iff_hlen;          /* local net header length */
    struct    uba_regs *iff_uba; /* uba regs, in vm */
    short    iff_flags;         /* used during uballoc's */
};
```

Additional structures are associated with each receive and transmit buffer, normally one each per interface; for read,

```
struct ifrw {
    caddr_t   ifrw_addr;        /* virt addr of header */
    short     ifrw_bdp;         /* unibus bdp */
    short     ifrw_flags;       /* type, etc. */
#define IFRW_W 0x01            /* is a transmit buffer */
    int       ifrw_info;        /* value from ubaalloc */
    int       ifrw_proto;       /* map register prototype */
    struct    pte *ifrw_mr;     /* base of map registers */
};
```

and for write,

```

struct ifxmt {
    struct    ifrw ifrw;
    caddr_t   ifw_base;           /* virt addr of buffer */
    struct    pte ifw_wmap[IF_MAXNUBAMR]; /* base pages for output */
    struct    mbuf *ifw_xtofree;   /* pages being dma'd out */
    short     ifw_xswapped;        /* mask of clusters swapped */
    short     ifw_nmr;             /* number of entries in wmap */
};
#define ifw_addr   ifrw.ifrw_addr
#define ifw_bdp    ifrw.ifrw_bdp
#define ifw_flags  ifrw.ifrw_flags
#define ifw_info   ifrw.ifrw_info
#define ifw_proto  ifrw.ifrw_proto
#define ifw_mr     ifrw.ifrw_mr

```

One of each of these structures is conveniently packaged for interfaces with single buffers for each direction, as follows:

```

struct ifuba {
    struct    ifubinfo ifu_info;
    struct    ifrw ifu_r;
    struct    ifxmt ifu_xmt;
};
#define ifu_uban   ifu_info.iff_uban
#define ifu_hlen   ifu_info.iff_hlen
#define ifu_uba    ifu_info.iff_uba
#define ifu_flags  ifu_info.iff_flags
#define ifu_w      ifu_xmt.ifrw
#define ifu_xtofree ifu_xmt.ifw_xtofree

```

The *ifubinfo* structure contains the general information needed to characterize the I/O-mapped buffers for the device. In addition, there is a structure describing each buffer, including UNIBUS resources held by the interface. Sufficient memory pages and bus map registers are allocated to each buffer upon initialization according to the maximum packet size and header length. The kernel virtual address of the buffer is held in *ifrw\_addr*, and the map registers begin at *ifrw\_mr*. UNIBUS map register *ifrw\_mr[-1]* maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifrw\_bdp*. The prototype of the map registers for read and for write is saved in *ifrw\_proto*.

When write transfers are not at least half-full pages on page boundaries, the data are just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is at least half a page long and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifw\_wmap* when the transfer completes. The mbufs containing the mapped pages are placed on the *ifw\_xtofree* queue to be freed after transmission.

When read transfers give at least half a page of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers; all use the structures described above.

```

if_ubaminit(ifubinfo, uban, hlen, nmr, ifr, nr, ifx, nx);
if_ubainit(ifuba, uban, hlen, nmr);

```

*if\_ubaminit* allocates resources on UNIBUS adapter *uban*, storing the information in the *ifubinfo*, *ifrw* and *ifxmt* structures referenced. The *ifr* and *ifx* parameters are pointers to arrays of *ifrw* and *ifxmt* structures whose dimensions are *nr* and *nx*, respectively. *if\_ubainit* is a simpler, backwards-compatible interface used for hardware with single buffers of each type. They are

called only at boot time or after a UNIBUS reset. One data path (buffered or unbuffered, depending on the *ifu\_flags* field) is allocated for each buffer. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if\_wubaput* below). If *if\_ubainit* is called with memory pages already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization are successful, 0 otherwise.

```
m = if_ubaget(ifubinfo, ifr, totlen, off0, ifp);
```

```
m = if_rubaget(ifuba, totlen, off0, ifp);
```

*if\_ubaget* and *if\_rubaget* pull input data out of an interface receive buffer and into an mbuf chain. The first interface passes pointers to the *ifubinfo* structure for the interface and the *ifrw* structure for the receive buffer; the second call may be used for single-buffered devices. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When the data amount to at least a half a page, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages, thus avoiding any copy. The receiving interface is recorded as *ifp*, a pointer to an *ifnet* structure, for the use of the receiving network protocol. A 0 return value indicates a failure to allocate resources.

```
if_wubaput(ifubinfo, ifx, m);
```

```
if_wubaput(ifuba, m);
```

*if\_ubaput* and *if\_wubaput* map a chain of mbufs onto a network interface in preparation for output. The first interface is used by devices with multiple transmit buffers. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Page-aligned data that are page-aligned in the output buffer are mapped to the UNIBUS in place of the normal buffer page, and the corresponding mbuf is placed on a queue to be freed after transmission. Any other mbufs which contained non-page-sized data portions are copied to the I/O space and then freed. Pages mapped from a previous output operation (no longer needed) are unmapped.

## 7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr\_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH      0      /* attach protocol */
#define PRU_DETACH      1      /* detach protocol */
#define PRU_BIND        2      /* bind socket to address */
#define PRU_LISTEN      3      /* listen for connection */
#define PRU_CONNECT     4      /* establish connection to peer */
#define PRU_ACCEPT      5      /* accept connection from peer */
#define PRU_DISCONNECT  6      /* disconnect from peer */
#define PRU_SHUTDOWN    7      /* won't send any more data */
#define PRU_RCVD        8      /* have taken data; more room now */
#define PRU_SEND        9      /* send this data */
#define PRU_ABORT       10     /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11     /* control operations on protocol */
#define PRU_SENSE       12     /* return status into m */
#define PRU_RCVOOB      13     /* retrieve out of band data */
#define PRU_SENDOOB     14     /* send out of band data */
#define PRU_SOCKADDR    15     /* fetch socket's address */
#define PRU_PEERADDR    16     /* fetch peer's address */
#define PRU_CONNECT2    17     /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO    18     /* 200ms timeout */
#define PRU_SLOWTIMO    19     /* 500ms timeout */
#define PRU_PROTORCV    20     /* receive from below */
#define PRU_PROTOSEND   21     /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[.pr_usrreq])(so, req, m, addr, rights);
int error; struct socket *so; int req; struct mbuf *m, *addr, *rights;
```

The mbuf data chain *m* is supplied for output operations and for certain other operations where it is to receive a result. The address *addr* is supplied for address-oriented requests such as *PRU\_BIND* and *PRU\_CONNECT*. The *rights* parameter is an optional pointer to an mbuf chain containing user-specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of the data mbuf chains on output operations. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

### PRU\_ATTACH

When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The “attach” request will always precede any of the other requests, and should not occur more than once.

### PRU\_DETACH

This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

### PRU\_BIND

When a socket is initially created it has no address bound to it. This request indicates that an address should be bound to an existing socket. The protocol module must verify that the requested address is valid and available for use.

### PRU\_LISTEN

The “listen” request indicates the user wishes to listen for incoming connection requests on the

associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A “listen” request always precedes a request to accept a connection.

#### PRU\_CONNECT

The “connect” request indicates the user wants to establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel81b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel80], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU\_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

#### PRU\_ACCEPT

Following a successful PRU\_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

#### PRU\_DISCONNECT

Eliminate an association created with a PRU\_CONNECT request.

#### PRU\_SHUTDOWN

This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown and/or notify a connected peer of the shutdown.

#### PRU\_RCVD

This request is made only if the protocol entry in the protocol switch table includes the PR\_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

#### PRU\_SEND

Each user request to send data is translated into one or more PRU\_SEND requests (a protocol may indicate that a single user send request must be translated into a single PRU\_SEND request by specifying the PR\_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

#### PRU\_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

#### PRU\_CONTROL

The “control” request is generated when a user performs a UNIX *ioctl* system call on a socket (and the *ioctl* is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention). The *rights* parameter contains a pointer to an *ifnet* structure if the *ioctl* operation pertains to a particular network interface.

#### PRU\_SENSE

The “sense” request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. This currently returns a standard *stat* structure. It typically contains only the optimal transfer size for the connection (based on buffer size, windowing



information and maximum packet size). The *m* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

#### PRU\_RCVOOB

Any “out-of-band” data presently available is to be returned. An mbuf is passed to the protocol module, and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf. An error may be returned if out-of-band data is not (yet) available or has already been consumed. The *addr* parameter contains any options such as MSG\_PEEK to examine data without consuming it.

#### PRU\_SENDOOB

Like PRU\_SEND, but for out-of-band data.

#### PRU\_SOCKADDR

The local address of the socket is returned, if any is currently bound to it. The address (with protocol specific format) is returned in the *addr* parameter.

#### PRU\_PEERADDR

The address of the peer to which the socket is connected is returned. The socket must be in a SS\_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

#### PRU\_CONNECT2

The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr\_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU\_SLOWTIMO).

#### PRU\_FASTTIMO

A “fast timeout” has occurred. This request is made when a timeout occurs in the protocol's *pr\_fastimo* routine. The *addr* parameter indicates which timer expired.

#### PRU\_SLOWTIMO

A “slow timeout” has occurred. This request is made when a timeout occurs in the protocol's *pr\_slowtimo* routine. The *addr* parameter indicates which timer expired.

#### PRU\_PROTORCV

This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

#### PRU\_PROTOSEND

This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked “addressed to protocol” instead of “addressed to user” are left to the protocol modules. No protocols currently use this facility.

## 8. Protocol/protocol interface

The interface between protocol modules is through the *pr\_usrreq*, *pr\_input*, *pr\_output*, *pr\_ctlinput*, and *pr\_ctloutput* routines. The calling conventions for all but the *pr\_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

### 8.1. *pr\_output*

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, “internet protocol control block”, passed between modules conveys per connection

state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on. UDP is based on the Internet Protocol, IP [Postel81a], as its transport. UDP passes a packet to the IP module for output as follows:

```
error = ip_output(m, opt, ro, flags);
int error; struct mbuf *m; *opt; struct route *ro; int flags;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is used in making routing decisions (and passing them back to the caller for use in subsequent calls). The final parameter, *flags* contains flags indicating whether the user is allowed to transmit a broadcast packet and if routing is to be performed. The broadcast flag may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occurred which could be detected immediately (no buffer space available, no route to destination, etc.).

## 8.2. pr\_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[]).pr_input(m, ifp);
struct mbuf *m; struct ifnet *ifp;
```

Each mbuf list passed is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the second parameter.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission. The software interrupt is enabled by the network interfaces when they place input data on the input queue.

## 8.3. pr\_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data).

The common calling convention for this routine is,

```
(void) (*protosw[]).pr_ctlinput(req, addr);
int req; struct sockaddr *addr;
```

The *req* parameter is one of the following,



```

#define PRC_IFDOWN          0      /* interface transition */
#define PRC_ROUTEDEAD       1      /* select new route if possible */
#define PRC_QUENCH          4      /* some said to slow down */
#define PRC_MSGSIZE         5      /* message size forced drop */
#define PRC_HOSTDEAD        6      /* normally from IMP */
#define PRC_HOSTUNREACH     7      /* ditto */
#define PRC_UNREACH_NET     8      /* no route to network */
#define PRC_UNREACH_HOST    9      /* no route to host */
#define PRC_UNREACH_PROTOCOL 10     /* dst says bad protocol */
#define PRC_UNREACH_PORT    11     /* bad port # */
#define PRC_UNREACH_NEEDFRAG 12     /* IP_DF caused drop */
#define PRC_UNREACH_SRCFAIL 13     /* source route failed */
#define PRC_REDIRECT_NET    14     /* net routing redirect */
#define PRC_REDIRECT_HOST   15     /* host routing redirect */
#define PRC_REDIRECT_TOSNET 14     /* redirect for type of service & net */
#define PRC_REDIRECT_TOSHST 15     /* redirect for tos & host */
#define PRC_TIMXCEED_INTRANS 18     /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS   19     /* lifetime expired on reass q */
#define PRC_PARAMPROB       20     /* header incorrect */

```

while the *addr* parameter is the address to which the condition applies. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol [Postel81c]), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

#### 8.4. pr\_ctloutput

This is the routine that implements per-socket options at the protocol level for *getsockopt* and *setsockopt*. The calling convention is,

```

error = (*protosw[l].pr_ctloutput)(op, so, level, optname, mp);
int op; struct socket *so; int level, optname; struct mbuf **mp;

```

where *op* is one of *PRCO\_SETOPT* or *PRCO\_GETOPT*, *so* is the socket from whence the call originated, and *level* and *optname* are the protocol level and option name supplied by the user. The results of a *PRCO\_GETOPT* call are returned in an mbuf whose address is placed in *mp* before return. On a *PRCO\_SETOPT* call, *mp* contains the address of an mbuf containing the option data; the mbuf should be freed before return.

### 9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases with which to be concerned, transmission of a packet and receipt of a packet; each will be considered separately.

#### 9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet \*), it transmits a fully formatted packet with the following call,

```

error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;

```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt

driven routine to actually transmit the packet. For unreliable media, such as the Ethernet, "successful" transmission simply means that the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are only those that can be detected immediately, and are normally trivial in nature (no buffer space, address format not handled, etc.). No indication is received if errors are detected after the call has returned.

## 9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In this system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued for the protocol module, and a VAX software interrupt is posted to initiate processing.

Three macros are available for queuing and dequeuing packets:

**IF\_ENQUEUE(*ifq*, *m*)**

This places the packet *m* at the tail of the queue *ifq*.

**IF\_DEQUEUE(*ifq*, *m*)**

This places a pointer to the packet at the head of queue *ifq* in *m* and removes the packet from the queue. A zero value will be returned in *m* if the queue is empty.

**IF\_DEQUEUEIF(*ifq*, *m*, *ifp*)**

Like IF\_DEQUEUE, this removes the next packet from the head of a queue and returns it in *m*.

A pointer to the interface on which the packet was received is placed in *ifp*, a (struct ifnet \*).

**IF\_PREPEND(*ifq*, *m*)**

This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF\_QFULL(*ifq*) returns 1 if the queue is filled, in which case the macro IF\_DROP(*ifq*) should be used to increment the count of the number of packets dropped, and the offending packet is dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);
} else
    IF_ENQUEUE(inq, m);
```

## 10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

### 10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rentry {
    u_long   rt_hash;           /* hash key for lookups */
    struct   sockaddr rt_dst;   /* destination net or host */
    struct   sockaddr rt_gateway; /* forwarding agent */
    short    rt_flags;          /* see below */
    short    rt_refcnt;         /* no. of references to structure */
    u_long   rt_use;            /* packets sent using route */
    struct   ifnet *rt_ifp;     /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g. DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). The first appropriate route discovered is used. By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to a "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (the desired final destination), a gateway to which to send the packet, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept, along with a count of "held references" to the dynamically allocated structure to insure that memory reclamation occurs only when the route is not in use. Finally, a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt\_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt\_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will contain the address of the eventual destination, while the local network header will address the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The *RTF\_GATEWAY* flag indicates that the route is to an "indirect" gateway agent, and that the local network header should be filled in from the *rt\_gateway* field instead of from the final internetwork destination address.

It is assumed that multiple routes to the same destination will not be present; only one of multiple routes, that most recently installed, will be used.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Current connections may be rerouted after notification of the



protocols by means of their *pr\_ctlinput* entries. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent “metrics” may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

## 10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure containing the desired destination:

```
struct route {
    struct      rtentry *ro_rt;
    struct      sockaddr ro_dst;
};
```

The route returned is assumed “held” by the caller until released with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit’s lifetime, while connection-less protocols, such as UDP, allocate and free routes whenever their destination address changes.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address, the new gateway to that destination, and the source of the redirect. Redirects are accepted only from the current router for the destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accessible from the host are ignored.

## 10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands *SIOCADDRT* and *SIOCDELRT* add and delete routing entries, respectively; the tables are read through the */dev/kmem* device. The decision to place policy decisions in a user process implies that routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

Several routing policy processes have already been implemented. The system standard “routing daemon” uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up-to-date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet EGP (Exterior Gateway Protocol), has been accomplished using a similar process.

## 11. Raw sockets

A raw socket is an object which allows users direct access to a lower-level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

### 11.1. Control blocks

Every raw socket has a protocol control block of the following form:

```
struct rawcb {
    struct rawcb *rcb_next;    /* doubly linked list */
    struct rawcb *rcb_prev;
    struct socket *rcb_socket; /* back pointer to socket */
    struct sockaddr rcb_faddr; /* destination address */
    struct sockaddr rcb_laddr; /* socket's address */
    struct sockproto rcb_proto; /* protocol family, protocol */
    caddr_t rcb_pcb;           /* protocol specific stuff */
    struct mbuf *rcb_options;  /* protocol specific options */
    struct route rcb_route;    /* routing information */
    short rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The *rcb\_proto* structure contains the protocol family and protocol number with which the raw socket is associated. The protocol, family and addresses are used to filter packets on input; this will be described in more detail shortly. If any protocol-specific information is required, it may be attached to the control block using the *rcb\_pcb* field. Protocol-specific options for transmission in outgoing packets may be stored in *rcb\_options*.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, *RAW\_LADDR* and *RAW\_FADDR*, indicate if a local and foreign address are present. Routing is expected to be performed by the underlying protocol if necessary.

### 11.2. Input processing

Input packets are “assigned” to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives unassigned packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
    struct sockproto raw_proto;
    struct sockaddr raw_dst;
    struct sockaddr raw_src;
};
```

and it is placed in a packet queue for the “raw input protocol” module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

- 1) The protocol family of the socket and header agree.
- 2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
- 3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
- 4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be

“block compared”.

### 11.3. Output processing

On output the raw *pr\_usrreq* routine passes the packet and a pointer to the raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

## 12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for “normal” network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be “turned off” as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

### 12.1. Memory management

The basic memory allocation routines manage a private page map, the size of which determines the maximum amount of memory that may be allocated by the network. A small amount of memory is allocated at boot time to initialize the mbuf and mbuf page cluster free lists. When the free lists are exhausted, more memory is requested from the system memory allocator if space remains in the map. If memory cannot be allocated, callers may block awaiting free memory, or the failure may be reflected to the caller immediately. The allocator will not block awaiting free map entries, however, as exhaustion of the page map usually indicates that buffers have been lost due to a “leak.” The private page table is used by the network buffer management routines in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple references to a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, it is copied or remapped into logically contiguous pages of memory from the network page pool if possible. Data smaller than half of the size of a page is copied into one or more 112 byte mbuf data areas.

### 12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the “silly window syndrome”

described in [Clark82] at both the sending and receiving ends.

### 12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

### 12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system attempts to generate a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

## 13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocol's prerogative to support larger-sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and is usually not stored in the socket's receive queue. A socket-level option, `SO_OOBINLINE`, is provided to force out-of-band data to be placed in the normal receive queue when urgent data is received; this sometimes ameliorates problems due to loss of data when multiple out-of-band segments are received before the first has been passed to the user. The `PRU_SENDOOB` and `PRU_RCVOOB` requests to the *pr\_usrreq* routine are used in sending and receiving data.

#### 14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page-sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without *a priori* knowledge of the format (e.g., by supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol* [Leffler84], places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion (in units of 512 bytes), and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct (
    short    protocol;    /* original protocol no. */
    short    length;      /* length of trailer */
);
```

The processing of the trailer protocol is very simple. On output, the local network header indicates that a trailer encapsulation is being used. The header also includes an indication of the number of data pages present before the trailer protocol header. The trailer protocol header is initialized to contain the actual protocol identifier and the variable length header size, and is appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate that as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.



## Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

## References

- [Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture*. Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.
- [BBN78] Bolt Beranek and Newman; Specification for the Interconnection of Host and IMP. BBN Technical Report 1822. May 1978.
- [Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.
- [Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP, RFC-813. Network Information Center, SRI International. July 1982.
- [DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture - General Description*. Order No. AA-K179A-TK. October 1980.
- [Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project - Implementation Description. Internetwork Working Group, IEN 168. January 1981.
- [ISO81] International Organization for Standardization. *ISO Open Systems Interconnection - Basic Reference Model*. ISO/TC 97/SC 16 N 719. August 1981.
- [Joy86] Joy, W.; Fabry, R.; Leffler, S.; McKusick, M.; and Karels, M.; Berkeley Software Architecture Manual, 4.3BSD Edition. *UNIX Programmer's Supplementary Documents*, Vol. 1 (PS1:6). Computer Systems Research Group, University of California, Berkeley. May, 1986.
- [Leffler84] Leffler, S.J. and Karels, M.J.; Trailer Encapsulations, RFC-893. Network Information Center, SRI International. April 1984.
- [Postel80] Postel, J. User Datagram Protocol, RFC-768. Network Information Center, SRI International. May 1980.
- [Postel81a] Postel, J., ed. Internet Protocol, RFC-791. Network Information Center, SRI International. September 1981.
- [Postel81b] Postel, J., ed. Transmission Control Protocol, RFC-793. Network Information Center, SRI International. September 1981.
- [Postel81c] Postel, J. Internet Control Message Protocol, RFC-792. Network Information Center, SRI International. September 1981.
- [Xerox81] Xerox Corporation. *Internet Transport Protocols*. Xerox System Integration Standard 028112. December 1981.
- [Zimmermann80] Zimmermann, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. Com-28(4); 425-432. April 1980.



## SENDMAIL — An Internetwork Mail Router

Eric Allman†

*Britton-Lee, Inc.  
1919 Addison Street, Suite 105.  
Berkeley, California 94704.*

### ABSTRACT

Routing mail through a heterogeneous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts as a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

*Sendmail* implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local

---

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@*a.cc.berkeley.arpa*" describes only the logical organization of the address space.

*Sendmail* is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

## 1. DESIGN GOALS

Design goals for *sendmail* include:

- (1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.
- (2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formatted addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.
- (3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.
- (4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.
- (5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.
- (6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias

file.

- (7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).
- (8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

## 2. OVERVIEW

### 2.1. System Organization

*Sendmail* neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley Mail, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission<sup>1</sup>. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

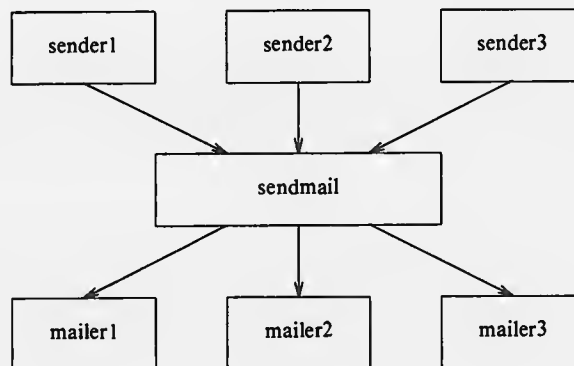


Figure 1 — Sendmail System Structure.

---

<sup>1</sup>except when mailing to a file, when *sendmail* does the delivery directly.

## 2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

### 2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

### 2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

### 2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2bsd IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a *sendmail* process on another machine.

## 2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

### 2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

*Sendmail* appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages delivered to the same recipient, as might occur if a person is in two groups.

### 2.3.2. Message collection

*Sendmail* then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

### 2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to *sendmail*. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

### 2.3.4. Queuing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

### 2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory<sup>2</sup>.

## 2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

## 2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling *sendmail* the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

---

<sup>2</sup>Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message — the "return to sender" function is always handled in one of these two ways.

### 3. USAGE AND IMPLEMENTATION

#### 3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

- (1) Anything in parentheses is thrown away (as a comment).
- (2) Anything in angle brackets ("`<>`") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form  
           user name <machine-address>  
 will send to the electronic "machine-address" rather than the human "user name."
- (3) Double quotes ( `"` ) quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently — for example, *user* and *"user"* are equivalent, but *\user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing<sup>3</sup>.

#### 3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e., not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("`|`") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("`/`") the name is used as a file name, instead of a login name.

Files that have *setuid* or *setgid* bits set but no *execute* bits set have those bits honored if *sendmail* is running as root.

#### 3.3. Aliasing, Forwarding, Inclusion

*Sendmail* reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

##### 3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

---

<sup>3</sup>Disclaimer: Some special processing is done after rewriting local names; see below.



### 3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

```
"|/usr/local/newmail myname"
```

will use a different incoming mailer.

### 3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

```
:Include: pathname
```

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

```
project: :include:/usr/project/userlist
```

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

### 3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

```
field-name: field-value
```

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

### 3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

### 3.6. Queued Messages

If the mailer returns a “temporary failure” exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

### 3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recompiled except

- (1) To change operating systems (V6, V7/32V, 4BSD).
- (2) To remove or insert the DBM (UNIX database) library.
- (3) To change ARPANET reply codes.
- (4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file “.mailcf” exists in the sender’s home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

#### 3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

#### 3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the “From:” and “Date:” lines.

Most configured headers will be automatically inserted in the outgoing message if they don’t exist in the incoming message. Certain headers are suppressed by some mailers.

#### 3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

#### 3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system,

except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcl!tef

might be mapped into:

tef@ucsfcl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

### 3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

## 4. COMPARISON WITH OTHER MAILERS

### 4.1. Delivermail

*Sendmail* is an outgrowth of *delivermail*. The primary differences are:

- (1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.
- (2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.
- (3) Forwarding and `:include:` features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).
- (4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.
- (5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.
- (6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

### 4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail*. For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code<sup>4</sup>.

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel<sup>5</sup> into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

#### 4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples<sup>6</sup>. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

### 5. EVALUATIONS AND FUTURE PLANS

*Sendmail* is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

*Sendmail* has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From"

<sup>4</sup>Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

<sup>5</sup>The MMDF equivalent of a *sendmail* "mailer."

<sup>6</sup>This is similar to the NBS standard.

lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August...") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

#### ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and *BerkNet* respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.



## REFERENCES

- [Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M.* 25, 4, April 82.
- [Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual*. R-2367-PAF. Rand Corporation. October 1979.
- [Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages*. RFC 733, NIC 41952. In [Feinler78]. November 1977.
- [Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System*. R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.
- [Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility — MMDF*. 6th Data Communication Symposium, Asilomar. November 1979.
- [Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*. RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.
- [Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM* 19, 7. July 1976.
- [Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook*. NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.
- [NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard*. Report No. ICST/CBOS 80-2. October 1980.
- [Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network*. RFC 542, NIC 17759. In [Feinler78]. August, 1973.
- [Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems*. Bell Laboratories. In *UNIX Programmer's Manual*, Seventh Edition, Volume 2. August, 1978.
- [Nowitz78b] Nowitz, D. A., *Uucp Implementation Description*. Bell Laboratories. In *UNIX Programmer's Manual*, Seventh Edition, Volume 2. October, 1978.
- [Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.
- [Postel77] Postel, J., *Mail Protocol*. NIC 29588. In [Feinler78]. November 1977.
- [Postel79a] Postel, J., *Internet Message Protocol*. RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.
- [Postel79b] Postel, J. B., *An Internetwork Message Structure*. In *Proceedings of the Sixth Data Communications Symposium*, IEEE. New York.



November 1979.

- [Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents*. RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.
- [Postel82] Postel, J. B., *Simple Mail Transfer Protocol*. RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.
- [Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network*. University of California, Berkeley California. 1979.
- [Shoens79] Shoens, K., *Mail Reference Manual*. University of California, Berkeley. In *UNIX Programmer's Manual*, Seventh Edition, Volume 2C. December 1979.
- [Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol*. RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.
- [Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.
- [Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications*. RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.
- [UNIX83] *The UNIX Programmer's Manual, Seventh Edition*, Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.



## On the Security of UNIX

*Dennis M. Ritchie*

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX<sup>†</sup> system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
    mkdir x
    cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories.

execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and-ed* with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login*, allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known,

but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the `crypt` command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user (*su*) command is writable, anyone can copy the shell onto it and get a password-free version of *su*. A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su*; other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.





## Password Security: A Case History

*Robert Morris*

*Ken Thompson*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

### INTRODUCTION

Password security on the UNIX<sup>†</sup> time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of employees, since they are not under any direct control and they may have intimate knowledge about

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories.

the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

## PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

## THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

## ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search*. Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length  $n$  that consist entirely of lower-case letters, the number of such passwords is  $26^n$ . If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than  $95^n$ . (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length  $n$  chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

$n$	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typically about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.



- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

- 15 were a single ASCII character;
- 72 were strings of two ASCII characters;
- 464 were strings of three ASCII characters;
- 477 were string of four alphameric;
- 706 were five letters, all upper-case or all lower-case;
- 605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

#### AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only  $2^{15}$  starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only  $2^{15}$  of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

#### IMPROVEMENTS TO THE FIRST APPROACH

##### 1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

##### 2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.



These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

### 3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 ( $2^{12}$ ). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

### 4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

### 5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

### CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will



likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

#### References

- [1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* 17 (July 1974), pp. 365-375.
- [2] *Proposed Federal Information Processing Data Encryption Standard*. Federal Register (40FR12134), March 17, 1975
- [3] Wilkes, M. V. *Time-Sharing Computer Systems*. American Elsevier, New York, (1968).
- [4] U. S. Patent Number 2,089,603.

## A Tour Through the Portable C Compiler

*S. C. Johnson*

AT&T Bell Laboratories

*Donn Seeley*

Department of Computer Science  
University of Utah

### ABSTRACT

Since its introduction, the Portable C Compiler has become the standard UNIX C compiler for many machines. Three quarters or more of the code in the compiler is machine independent and much of the rest can be generated easily using knowledge of the target architecture. This paper describes the structure and organization of the compiler and tries to further simplify the job of the compiler porter.

This document originally appeared with the Seventh Edition of UNIX, and has been revised and extended for publication with the Fourth Berkeley Software Distribution. The new material covers changes which have been made in the compiler since the Seventh Edition, and includes some discussion of secondary topics which were thought to be of interest in future ports of the compiler.

Revised April, 1986

### Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the DEC VAX, Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.<sup>1</sup>

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.<sup>2</sup>

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.<sup>3</sup> This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a



number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these machines, perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the other hand, a certain amount of the code requires hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

## Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.<sup>4</sup> One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.<sup>5</sup>

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the *#define* and *#include* statements, and related features (e.g., *#ifdef*, etc.). The two passes may optionally be followed by a machine dependent code improver.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The code improver, if used, converts the assembler code to more effective code, and the result is passed to the assembler. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

The division of the compiler into two passes is somewhat artificial. The compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes. This "one pass" compiler is the standard version on machines with large address spaces, such as the VAX.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees contain information useful to both logical passes. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

#### The Source Files

The compiler source consists of 25 source files. Several header files contain information which is needed across various source modules. *Manifest.h* has declarations for node types, type manipulation macros and other macros, and some global data definitions. *Macdefs.h* has machine-dependent definitions, such as the size and alignment of the various data representations. *Config.h* defines symbols which control the configuration of the compiler, including such things as the sizes of various tables and whether the compiler is "one pass". The compiler conditionally includes another file, *onepass.h*, which contains definitions which are particular to a "one pass" compiler. *Ndu.h* defines the basic tree building structure which is used throughout the compiler to construct expression trees. *Manifest.h* includes a file of opcode and type definitions named *pcclocal.h*; this file is automatically generated from a header file specific to the C compiler named *localdefs.h* and a public header file */usr/include/pcc.h*. Another file, *pcctokens*, is generated in a similar way and contains token definitions for the compiler's Yacc<sup>6</sup> grammar. Two machine independent header files, *pass1.h* and *pass2.h*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs.h*, contains declarations of register names, etc.

*Common.c* contains machine independent routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. This file can be compiled in two flavors, one for pass 1 and one for pass 2, depending on what conditional compilation symbol is used.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *cgram.y*, *code.c*, *common.c*, *local.c*, *optim.c*, *pftn.c*, *scan.c*, *stab.c*, *trees.c* and *xdefs.c*. *Scan.c* is the lexical analyzer, which provides tokens to the bottom-up parser which is defined by the Yacc grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Common.c* contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like. *Stab.c* contains machine dependent code for producing external symbol table information which can drive a symbolic debugger.

The second pass is produced by compiling and loading *allo.c*, *common.c*, *local2.c*, *match.c*, *order.c*, *reader.c* and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Common.c* defines certain service routines, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

### Data Structure Considerations

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest.h* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases. *Manifest.h* obtains some of its definitions from two other header files, *localdefs.h* and *pcc.h*. *Localdefs.h* contains definitions for operator symbols which are specific to the C compiler. *Pcc.h* contains definitions for operators and types which may be used by other compilers to communicate with a portable code generator based on pass 2; this code generator will be described later.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, -=), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *otype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number (=, +=, etc. ), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN,

and `ARY` represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus `PTR+INT` represents the type for an integer pointer, and

`ARY + (PTR<<2) + (FTN<<4) + DOUBLE`

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If  $t$  is a type word, `BTYPE(t)` gives the basic type. `ISPTR(t)`, `ISARY(t)`, and `ISFTN(t)` ask if an object of this type is a pointer, array, or a function, respectively. `MODTYPE(t,b)` sets the basic type of  $t$  to  $b$ . `DECREF(t)` gives the type resulting from removing the first operator from  $t$ . Thus, if  $t$  is a pointer to  $t'$ , a function returning  $t'$ , or an array of  $t'$ , then `DECREF(t)` would equal  $t'$ . `INCRF(t)` gives the type representing a pointer to  $t$ . Finally, there are operators for dealing with the unsigned types. `ISUNSIGNED(t)` returns true if  $t$  is one of the four basic unsigned types; in this case, `DEUNSIGN(t)` gives the associated 'signed' type. Similarly, `UNSIGNABLE(t)` returns true if  $t$  is one of the four basic types that could become unsigned, and `ENUNSIGN(t)` returns the unsigned analogue of  $t$  in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in `ndu.h`. The information stored for each pass is not quite the same; in the first pass, nodes contain dimension and size information, while in the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called `op`, containing the node number, and `type`, containing the type word. A function called `talloc()` returns a pointer to a new tree node. To free a node, its `op` field need merely be set to `FREE`. The other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, `left` and `right`, that contain pointers to the left and right descendants. Unary operator nodes have the `left` field, and a value field called `rval`. Leaf nodes, with no descendants, have two value fields: `lval` and `rval`.

At appropriate times, the function `tcheck()` can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function `tcopy(p)` takes a pointer  $p$  that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function `walkf(p,f)` performs a postorder walk of the tree pointed to by  $p$ , and applies the function  $f$  to each node. The function `fwalk(p,f,d)` does a preorder walk of the tree pointed to by  $p$ . At each node, it calls a function  $f$ , passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value  $d$  is the value passed down to the root. `Fwalk` is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

### Pass One

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

#### Lexical Analysis

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.

- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

### Parsing

As mentioned above, the parser is generated by Yacc from the grammar *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a `sizeof` construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.y* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values "on the side", independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In this way, when it is time to declare the name, the name and the type information can be quickly brought together. The "\$0" mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a `break` or `continue` statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enumeration declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

### Storage Classes

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this



transformation is machine dependent, and strongly depends on the storage class.

The classes include EXTERN (for externally declared, but not defined variables), EXTDEF (for external definitions), and similar distinctions for USTATIC and STATIC, UFORTRAN and FORTRAN (for fortran functions) and ULABEL and LABEL. The storage classes REGISTER and AUTO are obvious, as are STNAME, UNAME, and ENAME (for structure, union, and enumeration tags), and the associated MOS, MOU, and MOE (for the members). TYPEDEF is treated as a storage class as well. There are two special storage classes: PARAM and SNUL. SNUL is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, PARAM is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field; a  $k$  bit bit field has storage class  $k$  plus FIELD. This enables the size to be quickly recovered from the storage class.

### Symbol Table Maintenance

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block; in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if a formal parameter to a function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each \*, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enumerations) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the

symbol table *indx* in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to *auto*.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry "hidden", and making a new entry, marked "hiding". *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are "unhidden".

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a *goto* into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a "general purpose" field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in *dimtab* of the list of members. This list contains the symbol table indices for the structure members, terminated by a -1.

### Tree Building

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here; many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code

generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to handle issues which are unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *stref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary \* and &, structure assignment, and calls. In the case of unary \* and &, *buildtree* will cancel a \* applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is a rather strange area of the compiler (and of C!). The introduction of type casts only confounded this situation. Most of the conversion operators are generated by calls to *tymatch* and *ptmatch*, both of which are given a tree, and asked to make the operands agree in type. *Ptmatch* treats the case where one of the operands is a pointer; *tymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makety*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by *buildtree* is given to a machine dependent routine, *clocal*, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to *clocal* as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to *buildtree* may get confused about correct type of the subtrees; thus *clocal* is given the conversion operations only after the entire tree is built. This topic will be dealt with in more detail later.

### Initialization

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it is may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure; one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a ') has been seen for the current identifier).

When an initialization begins, the routine *beginit* is called; it handles the alignment restrictions, if any, and calls *instk* to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine *doinit* is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. *gotscal* is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; *endinit* cleans up. If a '{' or '}' is encountered in the string of initializers, it is handled by calling *ilbrace* or *irbrace*, respectively.

A central issue is the treatment of the "holes" that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, *inoff*, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine *inforce* which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines *fincode* (for floating point initialization), *incode* (for fields, and other initializations less than an int in size), and *cinit* (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token *STRING*, but does not do anything with the contents. The parser calls *getstr*, which sets up the appropriate location counters and flags, and calls *lxstr* to read and process the contents of the string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

## Statements

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot "fall through" the last statement.

Conditional branches are handled by generating an expression node, *CBRANCH*, whose left descendant is the conditional expression and the right descendant is an *ICON* node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, *FORCE*, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual

integer return value register.

### Optimization

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional; it must be called for proper operation of the compiler.

*Optim* is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of \* have been eliminated by *buildtree*.) The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true; on some machine, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$(x + a) + b$$

where *a* and *b* are constants, and attempts to combine *a* and *b* at compile time. A number of special cases are also examined; additions of 0 and multiplications by 1 are removed, although the correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion operator to keep the types from becoming fouled up. In cases where a relational operation is being done and one operand is a constant, the operands are permuted and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

### Machine Dependent Stuff

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcode* and *efcode*.

*Clocal* has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree*. There are two major areas where this is important: NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer; the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary \* node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number; a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different.

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal*, in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte



pointer to short or long pointer might require a shift or rotate operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

Another machine dependent issue in the first pass is the generation of external "symbol table" information. This sort of symbol table is used by programs such as symbolic debuggers to relate object code back to source code. Symbol table routines are provided in the file *stab.c*, which is included in the machine dependent sources for the first pass. The symbol table routines insert assembly code containing assembly pseudo-ops directly into the instruction stream generated by the compiler.

There are two basic kinds of symbol table operations. The simplest operation is the generation of a source line number; this serves to map an address in an executable image into a line in a source file so that a debugger can find the source code corresponding to the instructions being executed. The routine *psline* is called by the scanner to emit source line numbers when a nonempty source line is seen. The other variety of symbol table operation is the generation of type and address information about C symbols. This is done through the *outstab* routine, which is normally called using the FIX-DEF macro in the monster *defid* routine in *pftn.c* that enters symbols into the compiler's internal symbol table.

Yet another major machine dependent issue involves function prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.<sup>5</sup> The routine *bfcode* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfcode* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfcode* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfcode* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efcode* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return

value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called function is allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is non-reentrant. However, the function can now be called as a subroutine without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

#### First Pass Summary

There are many other issues which have been ignored here, partly to justify the title "tour", and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler's first pass the argument

`-X[flags]`

Some of the more interesting flags are `-Xd` for the defining and freeing of symbols, `-Xi` for initialization comments, and `-Xb` for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, `-Xddd` gives more information than `-Xd`. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

#### Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

#### Overview

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.



One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations. Most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

### The Machine Model

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs.h* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREGISTAREG for scratch AREG's, and SBREG and SBREGISTBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *istreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

### General Organization

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a *'*, and making note of the information about stack usage and register allocation contained on lines beginning with *]* and *[*. The expression trees, whose beginning is indicated by a line beginning with *.*, are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree; it attempts to delay some postfix *++* and *--* computations that might reasonably be done after the smoke clears. It also attempts to handle comma (',') operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only; don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG



(compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

*Codgen* first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *notoff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.<sup>7</sup> It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.<sup>8</sup> These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*; for example, when processing the operators &&, ||, and comma (','), that have a left to right evaluation, it is incorrect for *store* to examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the ? : operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is



made to find a match with a different cookie; for example, in order to compute an expression with cookie `INTMP` (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a `+` is a scratch register, the `+` can be replaced by a `+=` operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

$$a += b$$

where *a* and *b* are static variables.

To begin with, the whole expression tree is examined with cookie `FOREFF`, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the `+=` operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

$$a = a + b$$

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The machine dependent rule for `+` notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf *a*, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of *a* is rewritten in place to represent the register into which *a* is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

$$\text{reg} + b$$

to consider. Once again, there is no match, but the default rewriting rule rewrites the `+` as a `+=` operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

$$\text{reg} += b$$

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

$$a = \text{reg}$$

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different; *b* is loaded in to a register, and then an add to storage instruction generated to add this register in to *a*. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

### The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

If we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), then we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```

ASG PLUS,    INAREG,
              SAREG,    TINT,
              SNAME,    TINT,
                   0,    RLEFT,
                   "    add    AL,AR\n",

```

The top line specifies the operator (+) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim* to the output; upper case letters trigger various macro-like expansions. Thus, AL would expand into the Address form of the Left operand — presumably the register number. Similarly, AR would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```

ASG OPSIMP, INAREG|FORCC,
              SAREG,    TINT|TUNSIGNED|TPOINT,
              SAREG|SNAME|SOREG|SCON,    TINT|TUNSIGNED|TPOINT,
                   0,    RLEFT|RESCC,
                   "    OI    AL,AR\n",

```

Here, OPSIMP represents the operators +, -, !, &, and ^ . The OI macro in the output string expands into the appropriate Integer Opcode for the operator. The left and right sides can be integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

### The Template Matching Algorithm

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the operator of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the "left" descendant. If a node represents a unary operator or a leaf node (no descendants) the "right" descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees: STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator. These patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries; this may be of some importance on word addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and -1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to

search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

### Register Allocation

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered; the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.<sup>9</sup> The second issue is dealt with rather more successfully; a machine dependent function called *szty(t)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type *t*. If *szty* returns 2, an even/odd pair of A registers is allocated for each request. As part of its duties, the routine *usable* finds usable register pairs for various operations. This task is not as easy as it sounds; it does not suffice to merely use *szty* on the expression tree, since there are situations in which a register pair temporary is needed even though the result of the expression requires only one register. This can occur with assignment operator expressions which have int type but a double right hand side, or with relational expressions where one operand is float and the other double.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to an integer in it, we may load the integer into register 2 itself by saying:

```
L    2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR   3,3
IC   3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye; it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine *usable* to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible to have “double indexing” forms of addressing; these are represented by OREG’s with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results: the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error “cannot reclaim” is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

*Reclaim* also implements, in a curious way, C’s “usual arithmetic conversions”. When a value is generated into a temporary register, *reclaim* decides what the type and size of the result will be. Unless automatic conversion is specifically suppressed in the code template with the T macro, *reclaim* converts char and short results to int, unsigned char and unsigned short results to unsigned int, and float into double (for double only floating point arithmetic). This conversion is a simple type pun; no instructions for converting the value are actually emitted. This implies that registers must always contain a value that is at least as wide as a register, which greatly restricts the range of possible templates.

#### The Machine Dependent Interface

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi-Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here; that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

### The Rewriting Rules

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nextcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds; for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call *order* or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment operators, when rewritten, call the routine *setasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

$$a += b$$

to be rewritten as

$$a = a + b$$

This is a useful default for certain mixtures of strange types (for example, when *a* is a bit field and *b* an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

$$x ++$$

by

$$(x += 1) - 1$$

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of *x* is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREG's with a constant offset from a register, and *offstar* is called with a tree of the form

$$expr + const$$

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

### The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment operators. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problems have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where *A* and *B* are expressions; suppose further that *B* takes two registers, and *A* one. It is possible to compute the full expression in two registers by first computing *B*, and then, using the scratch register used by *B*, but not containing the answer, compute *A*. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

$$*(p + i)$$

where *p* is a pointer and *i* an integer, this can probably be done in one register on most machines.



Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both  $p$  and  $i$  into registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

### Register Allocation

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; *switch* and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

### Template Shortcuts

Some operations are just too hard or too clumsy to be implemented in code templates on a particular architecture.

One way to handle such operations is to replace them with function calls. The intermediate file reading code in *reader.c* contains a call to an implementation dependent macro MYREADER; this can be defined to call various routines which walk the code tree and perform transformations. On the VAX, for example, unsigned division and remainder operations are far too complex to encode in a template. The routine *hardops* is called from a tree walk in *myreader* to detect these operations and replace them with calls to the C runtime functions *udiv* and *urem*. (There are complementary functions *auidv* and *aurem* which are provided as support for unsigned assignment operator expressions; they are different from *udiv* and *urem* because the left hand side of an assignment operator expression must be evaluated only once.) Note that arithmetic support routines are always expensive; the compiler makes an effort to notice common operations such as unsigned division by a constant power of two and generates optimal code for these inline.

Another escape involves the routine *zzzcode*. This function is called from *expand* to process template macros which start with the character Z. On the VAX, many complex code generation problems are swept under the rug into *zzzcode*. Scalar type conversions are a particularly annoying issue; they are primarily handled using the macro ZA. Rather than creating a template for each possible conversion and result, which would be tedious and complex given C's many scalar types, this macro



allows the compiler to take shortcuts. Tough conversions such as unsigned into double are easily handled using special code under **ZA**. One convention which makes scalar conversions somewhat more difficult than they might otherwise be is the strict requirement that values in registers must have a type that is as wide or wider than a single register. This convention is used primarily to implement the "usual arithmetic conversions" of C, but it can get in the way when converting between (say) a char value and an unsigned short. A routine named *collapsible* is used to determine whether one operation or two is needed to produce a register-width result.

Another convenient macro is **ZP**. This macro is used to generate an appropriate conditional test after a comparison. This makes it possible to avoid a profusion of template entries which essentially duplicate each other, one entry for each type of test multiplied by the number of different comparison conditions. A related macro, **ZN**, is used to normalize the result of a relational test by producing an integer 0 or 1.

The macro **ZS** does the unlovely job of generating code for structure assignments. It tests the size of the structure to see what VAX instruction can be used to move it, and is capable of emitting a block move instruction for large structures. On other architectures this macro could be used to generate a function call to a block copy routine.

The macro **ZG** was recently introduced to handle the thorny issue of assignment operator expressions which have an integral left hand side and a floating point right hand side. These expressions are passed to the code generator without the usual type balancing so that good code can be generated for them. Older versions of the portable compiler computed these expressions with integer arithmetic; with the **ZG** operator, the current compiler can convert the left hand side to the appropriate floating type, compute the expression with floating point arithmetic, convert the result back to integral type and store it in the left hand side. These operations are performed by recursive calls to *zzzcode* and other routines related to *expand*.

An assortment of other macros finish the job of interpreting code templates. Among the more interesting ones: **ZC** produces the number of words pushed on the argument stack, which is useful for function calls; **ZD** and **ZE** produce constant increment and decrement operations; **ZL** and **ZR** produce the assembler letter code (l, w or b) corresponding to the size and type of the left and right operand respectively.

#### Shared Code

The *lint* utility shares sources with the portable compiler. *Lint* uses all of the machine independent pass 1 sources, and adds its own set of "machine dependent" routines, contained mostly in *lint.c*. *Lint* uses a private intermediate file format and a private pass 2 whose source is *lpass2.c*. Several modifications were made to the C scanner in *scan.c*, conditionally compiled with the symbol **LINT**, in order to support *lint*'s convention of passing "pragma" information inside special comments. A few other minor modifications were also made, e.g. to skip over *asm* statements.

The *f77* and *pc* compilers use a code generator which shares sources with pass 2 of the portable compiler. This code generator is very similar to pass 2 but uses a different intermediate file format. Three source files are needed in addition to the pass 2 sources. *fort.c* is a machine independent source file which contains a pass 2 main routine that replaces the equivalent routine in *reader.c*, together with several routines for reading the binary intermediate file. *fort.c* includes the machine dependent file *fort.h*, which defines two trivial label generation routines. A header file */usr/include/pcc.h* defines opcode and type symbols which are needed to provide a standard intermediate file format; this file is also included by the Fortran and Pascal compilers. The creation of this header file made it necessary to make some changes in the way the portable C compiler is built. These changes were made with the aim of minimizing the number of lines changed in the original sources. Macro symbols in *pcc.h* are flagged with a unique prefix to avoid symbol name collisions in the Fortran and Pascal compilers, which have their own internal opcode and type symbols. A *sed*(1) script is used to strip these prefixes, producing an include file named *pcclocal.h* which is specific to the portable C compiler and contains opcode symbols which are compatible with the original opcode symbols. A similar *sed* script is used to produce a file of Yacc tokens for the C grammar.

A number of changes to existing source files were made to accommodate the Fortran-style pass 2. These changes are conditionally compiled using the symbol FORT. Many changes were needed to implement single-precision arithmetic; other changes concern such things as the avoidance of floating point move instructions, which on the VAX can cause floating point faults when a datum is not a normalized floating point value. In earlier implementations of the Fortran-style pass 2 there were a number of stub files which served only to define the symbol FORT in a particular source file; these files have been removed for 4.3BSD in favor of a new compilation strategy which yields up to three different objects from a single source file, depending on what compilation control symbols are defined for that file.

The Fortran-style pass 2 uses a Polish Postfix intermediate file. The file is in binary format, and is logically divided into a stream of 32-bit records. Each record consists of an (*opcode, value, type*) triple, possibly followed inline by more descriptive information. The *opcode* and *type* are selected from the list in *pcc.h*; the *type* encodes a basic type, around which may be wrapped type modifiers such as "pointer to" or "array of" to produce more complex types. The function of the *value* parameter depends on the opcode; it may be used for a flag, a register number or the value of a constant, or it may be unused. The optional inline data is often a null-terminated string, but it may also be a binary offset from a register or from a symbolic constant; sometimes both a string and an offset appear.

Here are a few samples of intermediate file records and their interpretation:

Opcode	Type	Value	Optional Data	Interpretation
ICON	int	flag=0	binary=5	the integer constant 5
NAME	char	flag=1	binary=1, string="_foo_"	a character*1 element in a Fortran common block <i>foo</i> at offset 1
OREG	char	reg=11	offset=1, string="v.2-v.1"	the second element of a Fortran character*1 array, expressed as an offset from a static base register
PLUS	float			a single precision add
FTEXT		size=2	string=".text 0"	an inline assembler directive of length 2 (32-bit records)

### Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ...", or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in the code generator, and follow the tree through the process of being matched and rewritten. Some flags of



interest are `-e`, which prints the expression tree, `-r`, which gives information about the allocation of registers, `-a`, which gives information about the performance of *rallo*, and `-o`, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

### Compiler Extensions

The portable C compiler makes a few extensions to the language described by Ritchie.

*Single precision arithmetic.* "All floating arithmetic in C is carried out in double-precision; whenever a float appears in an expression it is lengthened to double by zero-padding its fraction." —Dennis Ritchie.<sup>1</sup> Programmers who would like to use C to write numerical applications often shy away from it because C programs cannot perform single precision arithmetic. On machines such as the VAX which can cleanly support arithmetic on two (or more) sizes of floating point values, programs which can take advantage of single precision arithmetic will run faster. A very popular proposal for the ANSI C standard states that implementations may perform single precision computations with single precision arithmetic; some actual C implementations already do this, and now the Berkeley compiler joins them.

The changes are implemented in the compiler with a set of conditional compilation directives based on the symbol SPRECC; thus two compilers are generated, one with only double precision arithmetic and one with both double and single precision arithmetic. The *cc* program uses a flag `-f` to select the single/double version of the compiler (*/lib/sgcom*) instead of the default double only version (*/lib/ccom*). It is expected that at some time in the future the double only compiler will be retired and the single/double compiler will become the default.

There are a few implementation details of the single/double compiler which will be of interest to users and compiler porters. To maintain compatibility with functions compiled by the double only compiler, single precision actual arguments are still coerced to double precision, and formal arguments which are declared single precision are still "really" double precision. This may change if function prototypes of the sort proposed for the ANSI C standard are eventually adopted. Floating point constants are now classified into single precision and double precision types. The precision of a constant is determined from context; if a floating constant appears in an arithmetic expression with a single precision value, the constant is treated as having single precision type and the arithmetic expression is computed using single precision arithmetic.

Remarkably little code in the compiler needed to be changed to implement the single/double compiler. In many cases the changes overlapped with special cases which are used for the Fortran-style pass 2 (*/lib/f1*). Most of the single precision changes were implemented by Sam Leffler.

*Preprocessor extensions.* The portable C compiler is normally distributed with a macro preprocessor written by J. F. Reiser. This preprocessor implements the features described in Ritchie's reference manual; it removes comments, expands macro definitions and removes or inserts code based on conditional compilation directives. Two interesting extensions are provided by this version of the preprocessor:

- When comments are removed, no white space is necessarily substituted; this has the effect of *re-tokenizing* code, since the PCC will reanalyze the input. Macros can thus create new tokens by clever use of comments. For example, the macro definition `"#define foo(a,b) a/**/b"` creates a macro *foo* which concatenates its two arguments, forming a new token.
- Macro bodies are analyzed for macro arguments without regard to the boundaries of string or character constants. The definition `"#define bar(a) "a\n"` creates a macro which returns the literal form of its argument embedded in a string with a newline appended.

These extensions are not portable to a number of other C preprocessors. They may be replaced in the future by corresponding ANSI C features, when the ANSI C standard has been formalized.

### Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

### Acknowledgements

I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun. —S. C. Johnson

Many people have contributed fixes and improvements to the current Berkeley version of the compiler. A number of really valuable fixes were contributed by Ralph Campbell, Sam Leffler, Kirk McKusick, Arthur Olsen, Donn Seeley, Don Speck and Chris Torek, but most of the bugs were spotted by the legions of virtuous C programmers who were kind enough to let us know that the compiler was broken and when the heck were we going to get it fixed? Thank you all. —Donn Seeley

### References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3
3. A. Snyder, *A Portable Compiler for the Language C*, Master's Thesis, M.I.T., Cambridge, Mass., 1974.
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104, January 1978.
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence*, 1977.
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.



7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.*, vol. 23, no. 3, pp. 488-501, 1975. Also in *Proc. ACM Symp. on Theory of Computing*, pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.*, vol. 17, no. 4, pp. 715-728, October 1970. Reprinted as pp. 229-247 in *Compiler Techniques*, ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages*, pp. 21-28, January 1977.

# Writing NROFF Terminal Descriptions

Eric Allman  
Britton-Lee, Inc.

## 1. INTRODUCTION

As of the Version 7 Phototypesetter release of UNIX,\* NROFF has supported terminal description files. These files describe the characteristics of available hard-copy printers. This document describes some of the details of how to write terminal description files.

*Disclaimer.* This document describes the results of my personal experience. The effects of changing some of the fields from the norms may not be well defined, even if it seems like it "ought" to work given the descriptions herein. These tables are known to vary slightly for different versions of UNIX. I have not seen UNIX 3.0 at this time, so this may be irrelevant in that context.

## 2. GENERAL

When NROFF starts up, it looks for a `-T` flag describing the terminal type. For example, if the command line is given as

```
nroff -T300s
```

NROFF prepares output for a *DTC300S* terminal. This terminal is described in the file `/usr/lib/term/tab300s` on most systems.

If no `-T` flag is given, the terminal type 37 (ASR 37 — a relic assumed for historical humor only) is assumed.

The terminal description table is a stripped ".o" file generated from a data structure, shown in figure one. This structure can be dealt with in two sections: the terminal capability descriptor (everything to `codetab`), and the output descriptor.

## 3. TERMINAL CAPABILITIES

The section of the data structure up to but excluding *codetab* describes the basic functions and setup requirements of the terminal. Distances are measured in "units," which are 1/240 of an inch in NROFF. In general, NROFF assumes that there is a "plot mode" on the terminal that allows you to move in small increments. A terminal has a resolution when in plot mode that is measured in units. This limits how well the terminal can simulate printing Greek and special characters.

### 3.1. bset, breset

These fields define bits in a vanilla `stty(2)` word (`sg_flags`) to set and clear respectively when NROFF starts. They are normally represented in octal, although you could include `<sgtty.h>`. [Note: these fields are presumably different in UNIX 3.0.]

---

\*UNIX is a trademark of Bell Laboratories.



---

```

#define INCH 240      /* one inch in units */
struct
{
    int bset;         /* stty bits to set */
    int breset;       /* stty bits to reset */
    int Hor;          /* horizontal resolution in units */
    int Vert;         /* vertical resolution in units */
    int Newline;      /* the distance a newline moves */
    int Char;         /* the distance one char moves */
    int Em;           /* size of an Em */
    int Halfline;     /* the distance a halfline up/down moves */
    int Adj;          /* default adjustment width */
    char *twinit;      /* string to init the terminal */
    char *twreset;     /* string to reset the terminal */
    char *twnl;       /* string to send a newline (CR-LF) */
    char *hlr;        /* half line reverse string */
    char *hlf;        /* half line forward string */
    char *flr;        /* full line reverse string */
    char *bdon;       /* string to turn boldface on */
    char *bdoff;      /* string to turn boldface off */
    char *ploton;     /* string to turn plot on */
    char *plotoff;    /* string to turn plot off */
    char *up;         /* move up in plot mode */
    char *down;       /* move down in plot mode */
    char *right;      /* move right in plot mode */
    char *left;       /* move left in plot mode */
    char *codetab[256-32]; /* the codes to send for characters */
    int zzz;         /* padding */
};

```

Figure 1 — the terminal descriptor data structure

---

### 3.2. Hor, Vert

These represent the horizontal and vertical resolution respectively of the terminal when it is in plot mode. They are given in units.

### 3.3. Newline

This field describes the distance that the *twnl* field (below) will move the paper; it is literally the size of a newline.

### 3.4. Char

This is the distance that a regular character will move the print head to the right.

### 3.5. Em

The “em” is a typesetting unit, approximately equal to the width of the letter “m”. In NROFF driver tables, this must be the distance a space or backspace character will move the carriage.



### 3.6. Halfline

This is the distance that the *hlf* or *hlf* strings move the print head (reverse or forward respectively).

### 3.7. Adj

This is the resolution that NROFF will normally adjust your lines to horizontally. Typically this is the same as Char. If the *-e* flag is given to NROFF, output resolution will be to the full device resolution.

### 3.8. twinit, twrest

These strings are output when NROFF starts and finishes respectively.

### 3.9. twnl

This string is output when NROFF wants to do a carriage return. Typically it will be “\r\n”. Remember, the terminal will normally have CRMOD turned off when this is set.

### 3.10. hlf, hlf

These strings are sent to move the carriage back or forward one half line respectively. The actual amount that they moved is defined by Halfline. The carriage should be left in the same column.

### 3.11. flr

The string to send to move a full line backwards. This should leave the carriage in the same column.

### 3.12. bdon, bdoft

These strings are sent to turn boldface mode on and off respectively. Normally this will set the terminal into overstrike mode. If they are not given, some newer versions of NROFF will output the characters four times to force overstriking.

### 3.13. ploton, plotoff

These strings turn plot mode on and off respectively. In plot mode, the carriage moves a very small amount, and only under specific control; i.e., characters do not automatically cause any carriage motion.

### 3.14. up, down, right, left

These strings are only output in plot mode. They should move the carriage up, down, left, and right respectively; they will move the carriage a distance of Hor or Vert as appropriate.

### 3.15. An Example

Consider the following table describing a DTC300S:

/*bset*/	0,
/*breset*/	0177420,
/*Hor*/	INCH/60,
/*Vert*/	INCH/48,
/*Newline*/	INCH/6,
/*Char*/	INCH/10,
/*Em*/	INCH/10,
/*Halfln*/	INCH/12,
/*Adj*/	INCH/10,
/*twinit*/	"\033\006",
/*twrest*/	"\033\006",
/*twini*/	"\015\n",
/*hlr*/	"\033H",
/*hlf*/	"\033h",
/*flr*/	"\032",
/*bdon*/	"",
/*bdoff*/	"",
/*ploton*/	"\006",
/*plotoff*/	"\033\006",
/*up*/	"\032",
/*down*/	"\n",
/*right*/	"",
/*left*/	"\b",

This describes a terminal that should have the ALLDELAY and CRMOD bits turned off, 1/60" horizontal and 1/48" vertical resolution, six lines per inch and ten characters per inch, including space, halfln takes 1/12" (one half of a full line), should send ESC-control-F to initialize and reset the terminal (to insure that it is in a normal state), takes <CR><LF> to give a newline, <ESC>H to move back one half line, <ESC>h to move forward one half line, control-Z to move back one full line, has no bold mode, takes control-F to enter plot mode and escape-control-F to exit plot mode, and uses control-Z, linefeed, space, and backspace to move up, down, right, and left respectively when in plot mode.

#### 4. CHARACTER DESCRIPTIONS

There is one character description for each possible character to be output. The easiest way to find what character corresponds to what position is to edit an existing character table; one is given in the appendix as an example. Character representations are represented as a string per character.

The first character of the string is interpreted as a binary number giving the number of character spaces taken up by this character. For regular characters this will always be "\001", but Greek and special characters can take more. If the 0200 bit is set in this character, it indicates that the character should be underlined if we are in italic (underline) mode. Thus, alphabetic and numeric descriptions will begin "\201".

The remainder of the string is output to represent the character. If the first output character (i.e., the second character in the total string) has the 0200 bit set, the character will be output in plot mode so that fancy characters can be built up from existing characters. If necessary, the "\200" character can be used as a null character to force NROFF to set the terminal into plot mode. All characters without the 0200 bit are output literally; characters with the 0200 bit are not output, but are used to indicate local carriage movement. The next two bits (0140 bits) represent direction:

```

0200 right
0240 left
0300 down
0340 up

```

The bottom five bits represent a distance in terminal resolution units. This is rather confusing, but the examples should make this much more clear.

#### 4.1. Some Examples

The following examples are from the DTC300S table:

```

"\001 ", /*space*/
"\001 =", /*=*/
"\201A", /*A*/

```

These entries show that all of these characters take one character width when output. The letter A is underlined in italic mode, but neither space nor equal sign is.

```

"\001o\b+", /*bullet*/
"\002[", /*square*/
"\202fi", /*fi*/

```

The bullet character takes only one character position, but is created by outputting the letter "o" and overstriking it with a plus sign. The square character is approximated with two brackets; it takes two full character positions when output. The "fi" ligature is produced using the letters "f" and "i" (surprise!); it is underlined in italic mode.

```

"\001\241c\202(\241", /*alpha*/
"\001\200B\242\302|\202\342", /*beta*/

```

The letters alpha and beta both take a single character position. The alpha is output by entering plot mode, moving left 1 terminal unit (1/60" if you recall), outputting the letter "c", moving right 2/60", outputting a left parenthesis, and finally moving left 1/60"; it is critical that the net space moved be zero both horizontally and vertically. The beta first has a dummy 0200 character to enter plot mode but not output anything. It then outputs a "B", moves left 2/60", moves down 2/48", outputs a vertical bar (which is designed to partially overstrike the left edge of the "B", and finally move right 2/60" and up 2/48" to set us back to the right place.

## 5. INSTALLATION

To install a terminal descriptor, make it up by editing an existing terminal descriptor. Assuming your terminal name is *term*, call your new descriptor *tabterm.c*. Then, execute the following commands:

```

cc -c tabterm.c
strip tabterm.c
cp tabterm.o /usr/lib/term/tabterm

```

The directory */usr/src/cmd/troff/term* typically has a shell file to do this.



## APPENDIX

### A Sample Table

This table describes the DTC 300S.

```
#define INCH 240
/*
DASI300S
nroff driving tables
width and code tables
*/

struct {
    int bset;
    int breset;
    int Hor;
    int Vert;
    int Newline;
    int Char;
    int Em;
    int Halfline;
    int Adj;
    char *twinit;
    char *twrest;
    char *twnl;
    char *h1r;
    char *h1f;
    char *flr;
    char *bdon;
    char *bdoff;
    char *ploton;
    char *plotoff;
    char *up;
    char *down;
    char *right;
    char *left;
    char *codetab[256-32];
    int zzz;
} t = {
/*bset*/0,
/*breset*/    0177420,
/*Hor*/      INCH/60,
/*Vert*/     INCH/48,
/*Newline*/  INCH/6,
/*Char*/     INCH/10,
/*Em*/       INCH/10,
/*Halfline*/ INCH/12,
/*Adj*/      INCH/10,
```

```

/*twinit*/      "\033\006".
/*twrest*/      "\033\006".
/*twnl*/        "\015\n".
/*hlr*/         "\033H".
/*hlf*/         "\033h".
/*flr*/         "\032".
/*bdon*/        "",
/*bdoff*/       "",
/*ploton*/      "\006".
/*plotoff*/     "\033\006".
/*up*/          "\032".
/*down*/        "\n".
/*right*/       " ",
/*left*/        "\b".
/*codetab*/
"\001 ", /*space*/
"\001!", /*!* */
"\001\"", /*** */
"\001#", /*## */
"\001$", /*$* */
"\001%", /*%* */
"\001&", /*&* */
"\001'", /*' close*/
"\001(", /*(* */
"\001)", /*)* */
"\001*", /*** */
"\001+", /*+* */
"\001,", /*,* */
"\001-", /*- hyphen*/
"\001.", /*.* */
"\001/", /*/* */
"\2010", /*0* */
"\2011", /*1* */
"\2012", /*2* */
"\2013", /*3* */
"\2014", /*4* */
"\2015", /*5* */
"\2016", /*6* */
"\2017", /*7* */
"\2018", /*8* */
"\2019", /*9* */
"\001:", /*:* */
"\001;", /*;* */
"\001<", /*<* */
"\001=", /*=* */
"\001>", /*>* */
"\001?", /*?* */
"\001@", /*@* */
"\201A", /*A* */
"\201B", /*B* */
"\201C", /*C* */
"\201D", /*D* */
"\201E", /*E* */

```



```

"\201F",      /*F*/
"\201G",      /*G*/
"\201H",      /*H*/
"\201I",      /*I*/
"\201J",      /*J*/
"\201K",      /*K*/
"\201L",      /*L*/
"\201M",      /*M*/
"\201N",      /*N*/
"\201O",      /*O*/
"\201P",      /*P*/
"\201Q",      /*Q*/
"\201R",      /*R*/
"\201S",      /*S*/
"\201T",      /*T*/
"\201U",      /*U*/
"\201V",      /*V*/
"\201W",      /*W*/
"\201X",      /*X*/
"\201Y",      /*Y*/
"\201Z",      /*Z*/
"\001[",      /*[*/
"\001\\",     /*\*/
"\001]",      /*]*/
"\001~",      /*~*/
"\001_",      /*_ dash*/
"\001'", /*' open*/
"\201a",      /*a*/
"\201b",      /*b*/
"\201c",      /*c*/
"\201d",      /*d*/
"\201e",      /*e*/
"\201f",      /*f*/
"\201g",      /*g*/
"\201h",      /*h*/
"\201i",      /*i*/
"\201j",      /*j*/
"\201k",      /*k*/
"\201l", /*l*/
"\201m",      /*m*/
"\201n",      /*n*/
"\201o",      /*o*/
"\201p",      /*p*/
"\201q",      /*q*/
"\201r",      /*r*/
"\201s",      /*s*/
"\201t",      /*t*/
"\201u",      /*u*/
"\201v",      /*v*/
"\201w",      /*w*/
"\201x",      /*x*/
"\201y",      /*y*/
"\201z",      /*z*/

```

```

"\001{", /*{*/
"\001|", /*|*/
"\001}", /*}*/
"\001~", /*~*/
"\000\0", /*narrow sp*/
"\001-", /*hyphen*/
"\001o\b+", /*bullet*/
"\002[]", /*square*/
"\001-", /*3/4 em*/
"\001_", /*rule*/
"\000\0", /*1/4*/
"\000\0", /*1/2*/
"\000\0", /*3/4*/
"\001-", /*minus*/
"\202fi", /*fi*/
"\202fl", /*fl*/
"\202ff", /*ff*/
"\203ffi", /*ffi*/
"\203ffl", /*ffl*/
"\000\0", /*degree*/
"\000\0", /*dagger*/
"\000\0", /*section*/
"\001'", /*foot mark*/
"\001'", /*acute accent*/
"\001'", /*grave accent*/
"\001_", /*underrule*/
"\001/", /*slash (longer)*/
"\000\0", /*half narrow space*/
"\001 ", /*unpaddable space*/
"\001\241c\202\241", /*alpha*/
"\001\200B\242\302|\202\342", /*beta*/
"\001\200)\201/\241", /*gamma*/
"\001\200o\342<\302", /*delta*/
"\001<\b-", /*epsilon*/
"\001\200c\201\301,\241\343<\302", /*zeta*/
"\001\200n\202\302|\242\342", /*eta*/
"\001O\b-", /*theta*/
"\001i", /*iota*/
"\001k", /*kappa*/
"\001\200\\\304\241'\301\241'\345\202", /*lambda*/
"\001\200u\242,\202", /*mu*/
"\001\241(\203/\242", /*nu*/
"\001\200c\201\301,\241\343c\241\301'\201\301", /*xi*/
"\001o", /*omicron*/
"\001\341-\303'\301'\343", /*pi*/
"\001\200o\242\302|\342\202", /*rho*/
"\001\200o\301\202\341\242", /*sigma*/
"\001\200t\301\202\243\201\341", /*tau*/
"\001v", /*upsilon*/
"\001o\b/", /*phi*/
"\001x", /*chi*/
"\001\200/\302\202'\244'\202\342", /*psi*/
"\001\241u\203u\242", /*omega*/

```



```

"\001\242|\202\343-\303\202\242", /*Gamma*/
"\001\242/\303-\204-\343\\242", /*Delta*/
"\001O\b=", /*Theta*/
"\001\242/\204\\242", /*Lambda*/
"\000\0", /*Xi*/
"\001\242[]\204[]\242\343-\303", /*Pi*/
"\001\200>\302-\345-\303", /*Sigma*/
"\000\0", /**/
"\001Y", /*Upsilon*/
"\001o\b[\b]", /*Phi*/
"\001\200[]-\302\202'\244'\202\342", /*Psi*/
"\001\200O\302\241-\202-\241\342", /*Omega*/
"\000\0", /*square root*/
"\000\0", /*terminal sigma*/
"\000\0", /*root en*/
"\001>\b_", /*>=*/
"\001<\b_", /*<=*/
"\001=\b_", /*identically equal*/
"\001-", /*equation minus*/
"\001=\b~", /*approx =*/
"\000\0", /*approximates*/
"\001=\b/", /*not equal*/
"\002->", /*right arrow*/
"\002<-", /*left arrow*/
"\001|\b~", /*up arrow*/
"\000\0", /*down arrow*/
"\001=", /*equation equal*/
"\001x", /*multiply*/
"\001/", /*divide*/
"\001+\b_", /*plus-minus*/
"\001U", /*cup (union)*/
"\000\0", /*cap (intersection)*/
"\000\0", /*subset of*/
"\000\0", /*superset of*/
"\000\0", /*improper subset*/
"\000\0", /* improper superset*/
"\002oo", /*infinity*/
"\001\200o\201\301'\241\341'\241\341'\201\301", /*partial derivative*/
"\001\242\\343-\204-\303/\242", /*gradient*/
"\001\200-\202\341,\301\242", /*not*/
"\001\200|\202'\243\306'\241'\202\346", /*integral sign*/
"\000\0", /*proportional to*/
"\000\0", /*empty set*/
"\000\0", /*member of*/
"\001+", /*equation plus*/
"\001r\bO", /*registered*/
"\001c\bO", /*copyright*/
"\001|", /*box rule */
"\001c\b/", /*cent sign*/
"\000\0", /*dbl dagger*/
"\000\0", /*right hand*/
"\001*", /*left hand*/
"\001**", /*math * */

```



```
"\000\0",    /*bell system sign*/
"\001|",     /*or (was star)*/
"\001O",     /*circle*/
"\001|",     /*left top (of big curly)*/
"\001|",     /*left bottom*/
"\001|",     /*right top*/
"\001|",     /*right bot*/
"\001|",     /*left center of big curly bracket*/
"\001|",     /*right center of big curly bracket*/
"\001|",     /*bold vertical*/
"\001|",     /*left floor (left bot of big sq bract)*/
"\001|",     /*right floor (rb of ")*/
"\001|",     /*left ceiling (lt of ")*/
"\001|");    /*right ceiling (rt of ")*/
```



## A Dial-Up Network of UNIX™ Systems

*D. A. Nowitz*

*M. E. Lesk*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

A network of over eighty UNIX<sup>†</sup> computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

### 1. Purpose

The widespread use of the UNIX system<sup>1</sup> within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back and forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.<sup>2,3</sup> Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories.



## 2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.<sup>4,5</sup> Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustrates the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

### 3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

#### File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways;

- a) by a system daemon,
- b) by one of the *uucp* or *uux* programs,
- c) by a remote system.

#### Scan For Work

The file names in the spool directory are constructed to allow the daemon programs (*uucico*, *uuxqt*) to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

#### Call Remote System

The call is made using information from several files which reside in the *uucp* program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

- [1] system name,
- [2] times to call the system (days-of-week and times-of-day),
- [3] device or device type to be used for call,
- [4] line speed,
- [5] phone number,
- [6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.



A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

### Line Protocol Selection

The remote system sends a message

*Pproto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

*Ucode*

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

### Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S	send a file,
R	receive a file,
C	copy complete,
H	hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE*'s spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

### Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

#### 4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usgldan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.<sup>6</sup> Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

#### 5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.



The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

## 6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.<sup>7</sup> There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

## 7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

## Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.



**References**

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.
2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2177-2200, 1978.
3. G. L. Chesson, "The Network UNIX System," *Operating Systems Review*, vol. 9, no. 5, pp. 60-66, 1975. Also in *Proc. 5th Symp. on Operating Systems Principles*.
4. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F, June 1974. IEEE Cat. No. 74CH0859-9-CSCB.
5. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56, February 1975.
6. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," *Comp. Sci. Tech. Rep. No. 41*, Bell Laboratories, Murray Hill, New Jersey, June 1976.
7. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.



S  
M  
M 21

## The Berkeley UNIX† Time Synchronization Protocol

Riccardo Gusella, Stefano Zatti, and James M. Bloom

Computer Systems Research Group  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA 94720

### Introduction

The Time Synchronization Protocol (TSP) has been designed for specific use by the program *timed*, a local area network clock synchronizer for the UNIX 4.3BSD operating system. *Timed* is built on the DARPA UDP protocol [4] and is based on a master slave scheme.

TSP serves a dual purpose. First, it supports messages for the synchronization of the clocks of the various hosts in a local area network. Second, it supports messages for the election that occurs among slave time daemons when, for any reason, the master disappears. The synchronization mechanism and the election procedure employed by the program *timed* are described in other documents [1,2,3].

Briefly, the synchronization software, which works in a local area network, consists of a collection of *time daemons* (one per machine) and is based on a master-slave structure. The present implementation keeps processor clocks synchronized within 20 milliseconds. A *master time daemon* measures the time difference between the clock of the machine on which it is running and those of all other machines. The current implementation uses ICMP *Time Stamp Requests* [5] to measure the clock difference between machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks.<sup>1</sup> It then sends to each *slave time daemon* the correction that should be performed on the clock of its machine. This process is repeated periodically. Since the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with synchronization. When a machine comes up and joins the network, it starts a slave time daemon, which will ask the master for the correct time and will reset the machine's clock before any user activity can begin. The time daemons therefore maintain a single network time in spite of the drift of clocks away from each other.

Additionally, a time daemon on gateway machines may run as a *submaster*. A submaster time daemon functions as a slave on one network that already has a master and as master on other networks. In addition, a submaster is responsible for propagating broadcast packets from one network to the other.

† UNIX is a trademark of AT&T Bell Laboratories.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, and by the Italian CSELT Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of CSELT.

<sup>1</sup> A clock is considered to be faulty when its value is more than a small specified interval apart from the majority of the clocks of the machines on the same network. See [1,2] for more details.

To ensure that service provided is continuous and reliable, it is necessary to implement an election algorithm that will elect a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned. Under our algorithm, slaves are able to realize when the master has stopped functioning and to elect a new master from among themselves. It is important to note that since the failure of the master results only in a gradual divergence of clock values, the election need not occur immediately.

All the communication occurring among time daemons uses the TSP protocol. While some messages need not be sent in a reliable way, most communication in TSP requires reliability not provided by the underlying protocol. Reliability is achieved by the use of acknowledgements, sequence numbers, and retransmission when message losses occur. When a message that requires acknowledgment is not acknowledged after multiple attempts, the time daemon that has sent the message will assume that the addressee is down. This document will not describe the details of how reliability is implemented, but will only point out when a message type requires a reliable transport mechanism.

The message format in TSP is the same for all message types; however, in some instances, one or more fields are not used. The next section describes the message format. The following sections describe in detail the different message types, their use and the contents of each field. NOTE: The message format is likely to change in future versions of timed.

### Message Format

All fields are based upon 8-bit bytes. Fields should be sent in network byte order if they are more than one byte long. The structure of a TSP message is the following:

- 1) A one byte message type.
- 2) A one byte version number, specifying the protocol version which the message uses.
- 3) A two byte sequence number to be used for recognizing duplicate messages that occur when messages are retransmitted.
- 4) Eight bytes of packet specific data. This field contains two 4 byte time values, a one byte hop count, or may be unused depending on the type of the packet.
- 5) A zero-terminated string of up to 256 ASCII characters with the name of the machine sending the message.

The following charts describe the message types, show their fields, and explain their usages. For the purpose of the following discussion, a time daemon can be considered to be in one of three states: slave, master, or candidate for election to master. Also, the term *broadcast* refers to the sending of a message to all active time daemons.

**Adjtime Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Adjustment			
Microseconds of Adjustment			
Machine Name			
...			

Type: TSP\_ADJTIME (1)

The master sends this message to a slave to communicate the difference between the clock of the slave and the network time the master has just computed. The slave will accordingly adjust the time of its machine. This message requires an acknowledgment.

**Acknowledgment Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_ACK (2)

Both the master and the slaves use this message for acknowledgment only. It is used in several different contexts, for example in reply to an Adjtime message.

**Master Request Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_MASTERREQ (3)

A newly-started time daemon broadcasts this message to locate a master. No other action is implied by this packet. It requires a Master Acknowledgment.



**Master Acknowledgement**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_MASTERACK (4)

The master sends this message to acknowledge the Master Request message and the Conflict Resolution Message.

**Set Network Time Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
...			

Type: TSP\_SETTIME (5)

The master sends this message to slave time daemons to set their time. This packet is sent to newly started time daemons and when the network date is changed. It contains the master's time as an approximation of the network time. It requires an acknowledgment. The next synchronization round will eliminate the small time difference caused by the random delay in the communication channel.

**Master Active Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_MASTERUP (6)

The master broadcasts this message to solicit the names of the active slaves. Slaves will reply with a Slave Active message.

**Slave Active Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_SLAVEUP (7)

A slave sends this message to the master in answer to a Master Active message. This message is also sent when a new slave starts up to inform the master that it wants to be synchronized.

**Master Candidature Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_ELECTION (8)

A slave eligible to become a master broadcasts this message when its election timer expires. The message declares that the slave wishes to become the new master.

**Candidature Acceptance Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_ACCEPT (9)

A slave sends this message to accept the candidature of the time daemon that has broadcast an Election message. The candidate will add the slave's name to the list of machines that it will control should it become the master.

**Candidature Rejection Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_REFUSE (10)

After a slave accepts the candidature of a time daemon, it will reply to any election messages from other slaves with this message. This rejects any candidature other than the first received.

**Multiple Master Notification Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_CONFLICT (11)

When two or more masters reply to a Master Request message, the slave uses this message to inform one of them that more than one master exists.

**Conflict Resolution Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_RESOLVE (12)

A master which has been informed of the existence of other masters broadcasts this message to determine who the other masters are.



**Quit Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_QUIT (13)

This message is sent by the master in three different contexts: 1) to a candidate that broadcasts an Master Candidature message, 2) to another master when notified of its existence, 3) to another master if a loop is detected. In all cases, the recipient time daemon will become a slave. This message requires an acknowledgement.

**Set Date Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
...			

Type: TSP\_SETDATE (22)

The program *date(1)* sends this message to the local time daemon when a super-user wants to set the network date. If the local time daemon is the master, it will set the date; if it is a slave, it will communicate the desired date to the master.

**Set Date Request Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
...			

Type: TSP\_SETDATEREQ (23)

A slave that has received a Set Date message will communicate the desired date to the master using this message.



**Set Date Acknowledgment Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_DATEACK (16)

The master sends this message to a slave in acknowledgment of a Set Date Request Message. The same message is sent by the local time daemon to the program *date(1)* to confirm that the network date has been set by the master.

**Start Tracing Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_TRACEON (17)

The controlling program *timedc* sends this message to the local time daemon to start the recording in a system file of all messages received.

**Stop Tracing Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_TRACEOFF (18)

*Timedc* sends this message to the local time daemon to stop the recording of messages received.



**Master Site Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_MSITE (19)

*Timedc* sends this message to the local time daemon to find out where the master is running.

**Remote Master Site Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_MSITEREQ (20)

A local time daemon broadcasts this message to find the location of the master. It then uses the Acknowledgement message to communicate this location to *timedc*.

**Test Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
( unused )			
( unused )			
Machine Name			
...			

Type: TSP\_TEST (21)

For testing purposes, *timedc* sends this message to a slave to cause its election timer to expire.  
NOTE: *timed* is not normally compiled to support this.



**Loop Detection Message**

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Hop Count	( unused )		
( unused )			
Machine Name			
...			

Type: TSP\_LOOP (24)

This packet is initiated by all masters occasionally to attempt to detect loops. All submasters forward this packet onto the networks over which they are master. If a master receives a packet it sent out initially, it knows that a loop exists and tries to correct the problem.

**References**

1. R. Gusella and S. Zatti, *TEMPO: A Network Time Controller for Distributed Berkeley UNIX System*, USENIX Summer Conference Proceedings, Salt Lake City, June 1984.
2. R. Gusella and S. Zatti, *Clock Synchronization in a Local Area Network*, University of California, Berkeley, Technical Report, to appear.
3. R. Gusella and S. Zatti, *An Election Algorithm for a Distributed Clock Synchronization Program*, University of California, Berkeley, CS Technical Report #275, Dec. 1985.
4. Postel, J., *User Datagram Protocol*, RFC 768. Network Information Center, SRI International, Menlo Park, California, August 1980.
5. Postel, J., *Internet Control Message Protocol*, RFC 792. Network Information Center, SRI International, Menlo Park, California, September 1981.

